MASTER EN SCIENCES INFORMATIQUES

Gestion mémoire pour les systèmes temps réel en Java

Mathu, Morgan

Award date:
2007

Awarding institution:
Universite de Namur

FUNDP
Institut d'informatique

Gestion mémoire pour les systèmes temps réel en Java

Mémoire présenté par
Morgan Mathu

Promoteur: Wim Vanhoof

Année académique 2006-2007

Abstract

Java is a high-level language that simplifies the programmer's task in building robust systems at the expense of a diminished performance and predictability, making it unsuitable for real-time programming.

The Java language lacks indeed serious capabilities for real-time programming since real-time system needs to respect some temporal constraints that the Java language isn't even able to express, i.e., the Java concurrency model isn't complete for real-time programming. Furthermore, this language uses garbage collection for memory management. The garbage collector can reduce the performances of an application and involves unpredictable pauses during program execution that are unacceptable in real-time systems.

This text addresses the issue of improving the memory management for real-time Java applications. It introduces the notions of memory management and garbage collection and explains the problems induced by garbage collection. The document also presents techniques to reduce pause times and improve the performance of garbage collectors.

Then, it explains the Java problems for real-time programming mentioned above. Finally, it presents improvements made to the Java platform's garbage collector since its beginning to stage with these problems and it concludes by looking into the RTSJ (Real-Time Specification for Java) which is an adapted version of the Java language suitable for real-time programming.

Résumé

Le Java est un langage de programmation de haut niveau qui simplifie la tâche tu programmeur dans le développement de systèmes robustes aux dépens des performances et de la prévisibilité, le rendant ainsi inapte pour la programmation en temps réel.

Le langage Java a en effet de sérieuses lacunes pour la programmation en temps réel étant donné que les systèmes en temps réel doivent respecter certaines contraintes temporelles que le langage Java n'est même pas capable d'exprimer, c'est-à-dire que le modèle de concurrence du Java n'est pas complet pour la programmation en temps réel. De plus, ce langage utilise la *garbage collection* comme système de gestion de la mémoire. Le *garbage collector* peut réduire les performances d'une application et entraîne des pauses imprévisibles de l'exécution du programme qui sont inacceptables dans des systèmes en temps réel.

Ce document traite de la question de l'amélioration de la gestion mémoire pour des applications Java en temps réel. Il introduit la notion de gestion mémoire et de *garbage collection* et explique les problèmes impliqués par la *garbage collection*. Le document présente aussi des techniques pour réduire les temps de pause et améliorer les performances des *garbage collectors*.

Ensuite, il explique les problèmes du Java pour la programmation en temps réel mentionnés ci-dessus. Finalement, il présente des améliorations faites au *garbage collector* de la plateforme Java depuis ses débuts pour palier à ces problèmes et termine en se penchant sur la RTSJ (*Real-Time Specification for Java*) qui est une version adaptée du langage Java apte à la programmation en temps-réel.

Remerciements

Je souhaite tout d'abord remercier mon promoteur Wim Vanhoof pour ses conseils avisés.

Un grand merci aussi à Aurore et à ma famille pour leur soutient durant mes années d'études et l'élaboration de ce document.

Table des matières

Introduction

Pour pouvoir correctement interagir en temps réel avec le monde qui les entoure, les systèmes informatiques en temps réel doivent être capables de répondre à certaines demandes en un temps limité et prévisible. La création de tels systèmes est rendue praticable grâce à la programmation concurrente qui permet d'exécuter plusieurs processus en parallèle plutôt que les uns à la suite des autres.

Cependant, la gestion du parallélisme entre différents processus implique un travail supplémentaire et peut avoir des comportements variant d'une exécution à une autre. En effet, les ressources physiques (comme par exemple le processeur) et logiques (comme des données en mémoire) d'un système informatique ne sont pas toujours disponibles aux mêmes moments d'une exécution à l'autre. De plus les requêtes venant du monde extérieur au système arrivent elles aussi à des moments relativement aléatoires.

Ce côté aléatoire peut faire varier le temps d'exécution de certaines tâches et peut donc être inacceptable au sein de certains systèmes en temps réel qui doivent pouvoir réagir en un temps maximum défini. Pour certains systèmes, appelés *hard real-time systems*, ne pas dépasser certaines échéances (*deadline*) est crucial pour leur bon fonctionnement. Pensons par exemple aux systèmes embarqués utilisés à bord d'un avion ou encore de machines d'assistance chirurgicales où le dépassement d'une *deadline* peut avoir de lourdes conséquences.

On distingue donc ces systèmes des *soft real-time systems* pour lesquels les *deadlines* sont importantes, mais qui peuvent continuer à fonctionner si les échéances sont manquées occasionnellement. Citons comme exemple un éditeur de textes.

Certains langages de programmation libèrent automatiquement la mémoire dont les programmes n'ont plus besoins grâce à un système appelé *garbage collection*, déchargeant ainsi le programmeur de cette tâche. Malheureusement, cette gestion automatique de la mémoire est bien souvent un des principaux problèmes lorsqu'on parle de programmation en temps réel. Le *garbage collector* est en effet un processus chargé de la libération de mémoire, mais il peut entraîner des pauses à des moments imprévisibles et pour des durées indéterminées, rendant ainsi le langage inapte à la programmation en temps réel.

Cependant, depuis ses débuts, la *garbage collection* a connu de nombreuses variantes et améliorations pour palier aux problèmes qu'elle entraînait.

Malheureusement malgré les différentes techniques trouvées au fil des ans, les *garbage collectors* ne fournissent pas des performances ou des garanties par rapport aux temps de réaction suffisantes.

Le langage Java est un de ces langages *garbage collected*. Il s'agit d'un premier problème qui l'empêche d'être un langage convenant pour la programmation en temps réel.

L'autre problème principal avec ce langage est que le modèle de concurrence du Java ne fournit pas les fonctionnalités (*facilities*) nécessaires pour pouvoir bien organiser et synchroniser les différentes tâches parallèles, appelées ici *threads*.

Même s'il est possible d'ajouter ces *facilities*, le *garbage collector* du Java empêche toujours ce langage de convenir aux systèmes en temps réel. Ce document a pour but d'étudier ce problème et est divisé en trois parties.

Il explique tout d'abord ce qu'est la gestion mémoire et approfondit la *garbage collection* et les problèmes qui en découlent.

Il nous montre ensuite des améliorations qui peuvent être apportées pour palier aux problèmes entrainés par la *garbage collection* en général.

Il montre enfin les problèmes qui rendent le langage Java inapte à la programmation de systèmes en temps réel et termine en expliquant les améliorations qui ont été faites pour palier aux problèmes du langage et tout particulièrement du *garbage collector*.

Gestion mémoire

1 Introduction

La plus ancienne forme de gestion de la mémoire est la gestion statique de la mémoire. Dans ce cas, toute la mémoire nécessaire aux variables et aux structures de données du programme est allouée statiquement par le programmeur ou le compilateur. Cette technique de gestion de la mémoire ne nécessite pas de temps au cours de l'exécution du programme. Elle possède cependant l'inconvénient évident de ne pas permettre de construire dynamiquement des structures de données en fonction des entrées du programme.

Or, un programme a généralement besoin d'allouer dynamiquement de la mémoire au cours de son exécution, ce qui est rendu possible par l'allocation dynamique de la mémoire. Les différents langages utilisant ce système se chargent en général de trouver de la place libre en mémoire à la demande du programmeur sans que celui-ci doive se soucier de la recherche d'une place libre: c'est le travail du *memory allocator* qui utilise généralement une structure de données appelée *heap* pour allouer de la place au cours de l'exécution d'un programme.

La quantité de mémoire disponible est physiquement limitée par le matériel. Il faut donc aussi pouvoir libérer la place prise par des données dont le programme n'aura plus besoin.

Des langages comme le C ou le C++ délèguent cette tâche au programmeur lui fournissant des instructions permettant de libérer de la mémoire qui a été allouée. Cependant, lors de la réalisation de systèmes de taille importante ou en temps réel, cette tâche devient vraiment complexe pour le programmeur et peut donc non seulement mener à une perte de temps relativement grande, mais aussi à des erreurs dans les programmes conçus, comme par exemple les *dangling pointers*: des pointeurs vers des endroits de la mémoire qui ont été libérés.

Dans d'autres langages, le système d'exécution va se charger de trouver automatiquement la mémoire qui peut être libérée au cours de l'exécution, ce qui est appelé "gestion de mémoire automatique". La technique utilisée pour parvenir à cette fin est la *garbage collection* (GC). Celle-ci fut au départ utilisée dans des langages déclaratifs ou fonctionnels comme le Haskell, le Lisp ou encore le Prolog. Elle est aussi utilisée dans des langages comme le Java ou le C# et consiste dans des langages orientés objet tels que ceux-ci à récupérer la place qui était occupée par des objets vers lesquels il n'y a plus de références.

Enfin, un problème qui est commun aux systèmes manuels et automatiques de gestion mémoire est la fragmentation. Au cours de l'utilisation de la mémoire avec des allocations et désallocations d'espaces de différentes tailles, la mémoire se fragmente peu à peu, c'est-à-dire qu'elle ne peut pas être totalement utilisée de façon continue à cause de petits espaces non alloués auxquels on trouve rarement une utilisation.

2 Gestion manuelle de la libération de la mémoire

Dans un langage comme le C par exemple, le programmeur va pouvoir utiliser une instruction qui va lui permettre d'allouer une zone mémoire libre de taille voulue. Le programmeur récupèrera donc un pointeur vers cette zone mémoire qu'il pourra utiliser à sa guise. Dans un langage orienté objet, une instruction permet de créer un objet dans une zone libre de la mémoire et de récupérer aussi une référence vers l'objet créé. Dans les deux cas, le programmeur va pouvoir utiliser et copier les références qu'il a obtenues comme il le souhaite.

Dans le cas de langages comme le C et le C++, le programmeur sera aussi chargé de libérer une zone mémoire qu'il a alloué lorsque le programme n'en n'a plus besoin. Il pourra faire cela en fournissant à une instruction une référence vers la zone mémoire à libérer.

Cette méthode de libération de la mémoire est relativement simple à implémenter et à exécuter. De plus, elle a l'avantage d'être prévisible (le programmeur décide quand libérer la mémoire) et bornée en temps (le temps pour libérer une zone mémoire est, au pire, proportionnel à la taille de cette zone, à moins qu'il ne faille recherche cette taille dans une liste, auquel cas il faut ajouter un temps qui est lui aussi borné et proportionnel à la longueur de cette liste).

Cependant, en plus d'être une charge supplémentaire pour le programmeur (qui peut s'avérer conséquente), une erreur de manipulation peut mener à des problèmes apparaissant au cours de l'exécution (*runtime errors*). Citons par exemple l'utilisation d'un pointeur après avoir libéré la zone mémoire qui lui correspond: *dangling pointers*. Ce type d'erreur peut arriver fréquemment si le système à développer est complexe et d'autant plus si celui-ci est *mutlithreaded*. A l'opposé, le programmeur peut aussi oublier de libérer certaines zones mémoire qui ne seront pourtant plus utilisées et provoquer ainsi des *memory leaks* ("fuites de mémoire") pouvant amener à la saturation de la mémoire.

Enfin, la façon dont le programmeur va libérer la mémoire est peut-être prévisible et bornée, mais elle n'est pas toujours optimale. En effet, un gestionnaire de mémoire automatique va pouvoir apporter certaines optimisations rendant le travail de gestion de la mémoire plus rapide, mais bien souvent au coût de place supplémentaire en mémoire et d'un manque de prévisibilité.

3 Gestion automatique de la libération de la mémoire

A l'opposé de la gestion manuelle, la seule tâche du programmeur ici est de demander de la mémoire lorsqu'il en a besoin. Un système de gestion automatique est mis en place dans le but de libérer les zones mémoire dont on est certain que le programme n'aura plus besoin. Le système pour réaliser cette tâche est connu sous le nom de *garbage collector* (GC).

Le principe de base des *garbage collectors* est de déterminer quels objets d'un programme ne seront plus utilisés et de récupérer les zones mémoire qu'ils utilisaient. Pour ce faire, le *memory allocator* et le *garbage collector* sont en général très liés, voire même confondus.

Il existe plusieurs variantes de *garbage collectors* qui diffèrent dans leur implémentation, mais aussi au niveau de leurs caractéristiques. Celles-ci nous sont expliquées dans les documents [2], [10], [17] et [21].

3.1 *Tracing Garbage Collectors*

Il s'agit ici du type le plus classique de *garbage collectors*. Pour déterminer si un objet ne sera plus utilisé, ils vont vérifier qu'il n'est plus atteignable (*reachable*), c'est-à-dire qu'il n'y a plus de références vers cet objet.

Un objet est atteignable s'il existe au moins une référence vers cet objet. Un objet peut être atteignable de deux manières. Il existe tout d'abord un ensemble d'objets atteignables que l'on peut appeler la racine ("*root set*"). Le *root set* est l'ensemble des objets référencés depuis la *stack* (servant à gérer les appels de méthodes) ou depuis des variables globales. Ensuite, un objet est aussi atteignable s'il est référencé par un objet atteignable.

Notons qu'un objet qui ne sera plus utilisé, à savoir un *"garbage"* n'est pas synonyme de *unreachable*. En effet, les *garbages* sont ceux qui ne sont plus utilisés. Un *garbage* peut donc encore être *reachable*, même si il ne sera plus utilisé. Cependant, on utilise cette notion de *reachable* car il est impossible de déterminer si un objet ne sera plus utilisé lorsqu'il est encore référencé. On distingue les deux notions en appelant un "vrai *garbage*" *semantic garbage* tandis qu'un objet *unreachable* est appelé *syntactic garbage*.

3.1.1 Algorithme de base: *tri-colour marking*

La libération des objets en mémoire se déroule en cycles de *garbage collection*. Un cycle démarre quand le collecteur le décide ou qu'il est notifié (par exemple si il reste peu de mémoire libre). L'algorithme de base utilise une technique appelée le marquage à trois couleurs (*tri-colour marking*) qui, comme son nom l'indique assigne une couleur à chaque objet: blanc, gris ou noir.

L'algorithme commence par colorer les objets du *root set* en gris, le reste des objets en blanc et l'ensemble des objets noirs est donc vide.

Il va ensuite parcourir l'ensemble des objets gris.
Pour chaque objet gris, il va tout d'abord passer en revue toutes les références qu'il contient et colorer en gris tous les objets blancs référencés. Ensuite, une fois que toutes les références de l'objet ont été parcourues, ce dernier est noirci.

Les objets noirs sont donc les objets qui ont été parcourus et qui ne référencent donc plus aucun objet blanc. Notons que dans certaines implémentations, l'ensemble de ces objets n'est pas vide au départ mais contient des objets pour lesquels ont peut facilement prouver qu'ils ne référencent pas des objets de l'ensemble blanc.

Une fois que tous les objets gris ont été colorés en noir, tous les objets *reachable* ont été parcourus et noircis. L'algorithme préserve comme invariant qu'aucun objet noir ne référence un objet blanc. Les objets blanc restant ne sont donc référencés par aucun des objets atteignables (noir) et sont donc des *garbages*.

Ce parcours des objets atteignables est le "*tracing*". Une fois cette partie de l'algorithme terminée, celui-ci se termine en libérant la mémoire occupée par les objets blancs identifiés comme *garbages*.

Différentes décisions de design peuvent être prises quant à l'implémentation du *garbage collector*.

3.1.2 *Moving* ou *non-moving strategy*

Tout d'abord, après avoir libéré l'espace mémoire des objets *unreachable*, le *garbage collector* peut soit laisser la mémoire dans l'état où elle est, soit recopier les objets restants (qui sont encore référencés) dans une autre zone mémoire (en mettant à jour les références vers ces objets). Ces deux stratégies sont respectivement appelées *non-moving* et *moving strategies*.

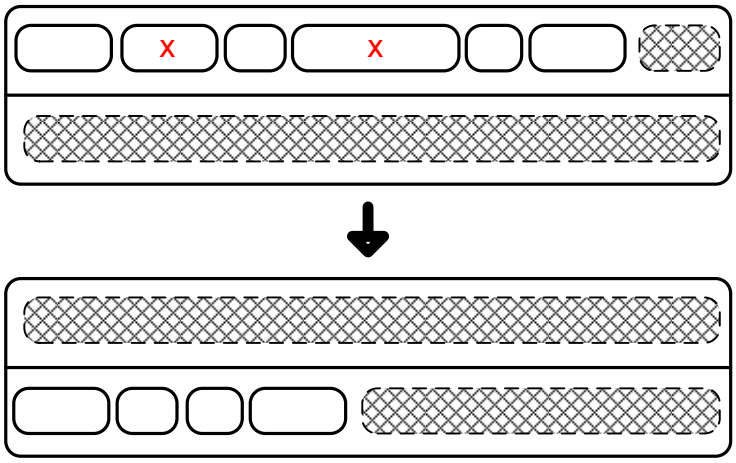
Le travail supplémentaire qu'entraîne la *moving strategy* peut en effet fournir les avantages suivants:
- une fois qu'une région a été libérée des objets *unreachable* et que les objets *reachable* ont été copiés, la région entière peut être considérée comme libre alors qu'autrement il faudrait supprimer les objets un par un en notant quelles zones sont désormais disponibles.
- de nouveaux objets peuvent être alloués plus rapidement. Etant donné que de longues régions en mémoire sont rendues disponibles, la recherche du *memory allocator* dans cette zone sera équivalente à la simple incrémentation d'un pointeur. Dans le cas contraire, les allocations et désallocations d'objets successives peuvent mener à un *heap* fortement fragmenté (entraînant des recherches dans des listes de zones libres pouvant prendre du temps).
- si les copies d'objets sont effectuées de façon optimale, des objets qui se référencent fréquemment peuvent être copiés dans une zone mémoire commune,

augmentant la probabilité qu'ils soient par exemple chargés ensemble dans une même page en mémoire cache.

3.1.3 *Copying* ou *mark-and-sweep*

Une autre distinction peut être faite entre les *tracing garbage collectors* suivant leur façon de retenir les trois ensembles (de couleur) durant la collection.

Une méthode utilisée pour cela est le *copying*. Elle consiste à partitionner la mémoire en deux espaces appelés *"from space"* et *"to space"*. Les objets sont alloués dans le *to space*. Lorsqu'une collection commence, les noms des deux espaces sont inversés. Au début de la collection, le *root set* est déplacé dans le (nouveau) *to space* et forme l'ensemble des objets gris au départ. L'ensemble blanc est constitué des objets dans le (nouveau) *from space*. Ensuite, le *to space* est parcouru et les objets *reachable* sont peu à peu déplacés à leur tour vers le *to space*, jusqu'à ce que tous les objets atteignables dans le *from space* aient été déplacés (l'ensemble noir étant constitué des objets du *to space* qui ont déjà été parcourus). Un fois tous les objets *reachable* déplacés, la mémoire occupée par le *from space* peut être libérée. C'est donc une *moving strategy* qui est utilisée ici. Ensuite, quand le programme reprend, il alloue toujours les objets dans le *"to space"* jusqu'à ce qu'il soit plein et que le processus soit répété. Cette approche est simple et évite le problème de fragmentation mais a le désavantage de pouvoir requérir un grand espace continu en mémoire.

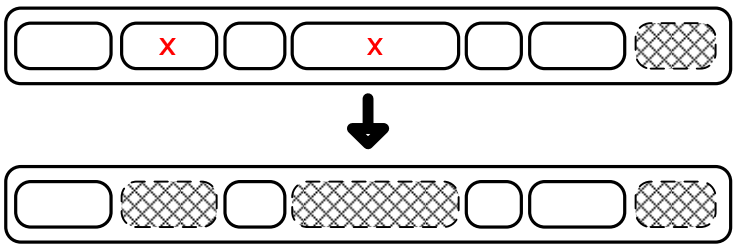


Copying

Un *garbage collector "mark-and-sweep"* retiendra quant à lui si un objet est blanc ou noir grâce à un bit contenu dans les objets. Les objets gris peuvent être connus grâce à un bit supplémentaire dans les objets ou grâce à une liste chaînée.

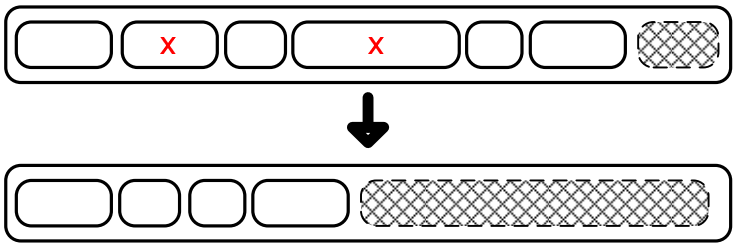
Cette méthode a pour avantage de permettre une *moving* ou une *non-moving strategy*, et ce même au moment de l'exécution suivant l'état de la mémoire. Notons que même si une *non-moving strategy* entraîne des problèmes de fragmentation, elle est nécessaire lorsque le *garbage collector* doit travailler avec des pointeurs conservatifs (cfr. point 3.1.6).

L'inconvénient de cette méthode est bien sur d'ajouter une petite quantité de données aux objets.



Mark-and-sweep (non-moving strategy)

Une méthode appelée *mark-and-compact* ou *mark-sweep-compact* est une approche se situant entre le *copying* et le *mark-and-sweep*. Cette technique marque les objets comme le fait un *mark-and-sweep garbage collector*, mais lors de la seconde phase, à la place de supprimer les objets *unreachable* ou de déplacer les objets *reachable* dans une autre zone, les objets *reachable* sont simplement compactés au début de la zone mémoire.



Mark-and-compact

3.1.4 Exécution du *garbage collector* par rapport au programme

La façon dont la *garbage collection* s'exécute par rapport au programme est aussi quelque chose qui varie d'un *garbage collector* à un autre. La méthode la plus évidente est appelé *stop-the-world* et consiste à arrêter l'exécution du programme le temps d'un cycle de *garbage collection*. Le fait d'arrêter le programme est l'inconvénient évident de cette méthode appelée aussi *batch garbage collection*. La méthode incrémentale quant à elle permet d'entrelacer l'exécution de la *garbage collection* avec l'exécution principale du programme (appelé *mutator*). Un tel système entraîne un charge de travail supplémentaire pour éviter les interférences entre les deux exécutions (modifications de références, création de nouveaux objets,…). Enfin, la méthode concurrente permet d'exécuter les deux parallèlement

sur un système multiprocesseur, mais ici aussi il est nécessaire d'éviter les interférences.

3.1.5 Introduction aux *garbage collectors* générationnels

Certains *garbage collector* séparent l'ensemble des objets en différentes générations qui sont groupées en mémoire dans beaucoup d'implémentations: ils sont appelés *generational* ou *ephemeral garbage collectors*. Il a en effet été empiriquement observé que dans la plupart des programmes, les objets les plus récemment créés sont ceux qui deviennent le plus rapidement *unreachable*. Dès lors, le *garbage collector* ne mettra qu'une partie des générations d'objets dans l'ensemble blanc (plus souvent les jeunes générations) lors de la plupart des cycles et seuls ceux-ci seront donc candidats à la *collection*. De plus, le système d'exécution garde quelques informations supplémentaires sur les références entre les générations que le *garbage collector* pourra utiliser pour prouver que certains objets sont *unreachable* sans devoir pour autant parcourir toutes les références de tous les objets en mémoire. Notons cependant qu'il s'agit ici d'une méthode heuristique et que certains objets *unreachable* ne seront pas collectés lors d'un cycle. Il est donc nécessaire d'exécuter occasionnellement un cycle de *garbage collection* complet (sur l'ensemble de la mémoire).

3.1.6 Les pointeurs: précis, conservatifs, et internes

Certains *garbage collectors* peuvent identifier correctement tous les pointeurs se trouvant en mémoire. Ils sont appelés "*precise collectors*" (ou "*exact collectors*" ou encore "*accurate collectors*").

D'autres *collectors* à l'opposé considèrent que n'importe quel champ en mémoire qui a le même format qu'un pointeur peut être un pointeur. Ceux-ci sont alors dits "*conservative*" (ou "*partly conservative*"). Il peut donc arriver que ces *collectors* ne libèrent pas certaines zones mémoires à cause de "faux pointeurs", mais cet inconvénient est rarement significatif en pratique. De plus, l'intérêt de la *conservative collection* est de pouvoir utiliser la *garbage collection* avec des langages qui ne l'avaient pas prévu et qui ne savent pas distinguer un pointeur en mémoire. Cependant, la *conservative collection* a quand même un inconvénient significatif: elle ne permet pas d'utiliser une *moving strategy* étant donnés qu'il faut pouvoir mettre à jour les pointeurs vers un objet qui est déplacé. En faisant cela avec un *conservative garbage collector*, on risquerait de modifier des zones mémoire qui ne sont pas des pointeurs.

Les *conservative collectors* sont donc souvent de type *mark-and-sweep*.

Cependant, on peut permettre le déplacement d'objets avec un *conservative collector* en référençant tout les objets via une indirection. Le seul pointeur vers un objet est alors son indirection ou "*handle*". Toute référence à un objet pointe dès lors vers son *handle* qui sert de relais. Cette technique fournit bien entendu une charge de travail supplémentaire et nécessite de la place ne mémoire pour les *handles*.

Un compromis entre les *collectors* précis et conservatifs sont les "*semi/partly conservative/accurate collectors*". Il s'agit d'une variante des

conservative collectors qui gère aussi les références exactes. Par exemple, les références du *root set* peuvent ne pas être *accurate* alors que celles des objets sur le *heap* le sont et peuvent donc être scannés de façon précise. De telles *garbage collectors* utilisent une approche hybride entre la *mark-and-sweep* et la *copying garbage collection.*

Enfin, certains langages autorisent des pointeurs "internes" à un objet: des pointeurs vers des champs dans l'objet. Si c'est le cas, il peut donc y avoir plusieurs adresses différentes qui référencent le même objet, ce qui complique encore le fait de déterminer si un objet est ou n'est pas un *garbage*.

3.2 Commentaires

Les différentes techniques de gestion citées apportent chacune une charge de travail supplémentaire au programme.

La désallocation manuelle nécessite la recherche d'une zone mémoire libre lors de l'allocation de mémoire. De plus, elle doit tenir à jour une liste des emplacements mémoire libres et doit gérer les problèmes de fragmentation. C'est aussi le cas pour certains *garbage collectors*, mais la *garbage collection* peut se libérer de ces charges en utilisant des techniques comme le *copying* ou le *mark-and-compact*.

Les *garbage collector* ont comme charge commune de devoir localiser les objets *reachable*. De plus, un *moving collector* devra aussi copier les objets *reachable* dans une autre zone mémoire. La méthode incrémentale devra quant à elle utiliser un mécanisme de barrière pour éviter les interférences avec le programme. Enfin, les *non-moving collectors* devront rechercher des zones mémoire libres et tenir à jour une liste des espaces libres, comme le système manuel.

3.3 *Reference counting*

En contraste avec la *tracing garbage collection*, le *reference counting* est une méthode où chaque objet possède un compteur du nombre de références pointant vers lui. Ce compteur est bien sur incrémenté quand une référence vers l'objet est créée et décrémenté quand une référence est détruite, écrasée ou est contenue dans un objet qui est collecté. Quand le compteur tombe à zéro, la mémoire occupée par l'objet peut être récupérée.

Cette méthode possède cependant deux inconvénients majeurs:
- quand deux ou plusieurs objets se référencent les uns les autres, les références peuvent créer un cycle empêchant les objets d'être désalloués car aucun des compteurs n'est à zéro. Un algorithme de détection de cycles est donc nécessaire pour ce genre de cas.
- lors de l'utilisation de cette technique dans un système *multithreaded*, les accès aux compteurs de références doivent être protégés par des systèmes de *locks*,

ce qui implique donc des ralentissements supplémentaires à ceux qui sont déjà engendrés par les accès aux compteurs.

Le *reference counting* est donc plus utile dans des situations où l'on peut garantir qu'il n'y aura pas de boucles ou encore dans des système avec une mémoire restreinte où l'on préfère donc réclamer les *garbages* le plus rapidement possible au détriment des performances.

Différentes variantes de *reference counters* sont expliquées dans le document [23].

3.3.1 *Deferred reference counting*

Les performances du comptage de références peuvent être améliorées si l'on ne tient pas compte de toutes les références. En effet, ne pas devoir accéder à chaque fois au compteur de référence réduit la surcharge de travail. Une technique connue sous le nom de *deferred reference counting* ne tient compte que des références issues des objets et ignore les références issues des variables du programme (comme par exemple la *stack*). Comme la plupart des références vers des objets proviennent des variables locales dans bon nombre de cas, ignorer ces références peut considérablement réduire la surcharge de travail due aux accès aux compteurs.

En agissant de la sorte, un objet ne peut bien entendu pas être collecté dès que son compteur tombe à zéro car il peut toujours être référencé par des variables du programme. A la place, cet objet est ajouté à la "*zero count table*". La réclamation des *garbages* se fait périodiquement en scannant les variables du programme et en collectant les objets listés dans cette table qui ne sont pas référencés par ces variables.

Le désavantage de cette méthode par rapport au comptage de références simple est que la réclamation des *garbages* n'est plus immédiate mais qu'il y a maintenant un temps de réaction. La rapidité à réagir est donc limitée par la fréquence à laquelle on scanne les variables du programme.

3.3.2 *Limited-field reference counting*

La taille du champ servant à compter les références peut être (trop) limitée. En effet, le champ peut être trop court pour pouvoir stocker le nombre de références maximum qui peuvent pointer vers un objet. On parle alors de *limited-field reference counting*. Il a en effet été observé que la plupart des objets ne sont pas référencés de nombreuses fois et certains systèmes en profitent pour réduire la taille que peut avoir le compteur de références. Lorsqu'un objet atteint le maximum de références que peut contenir son compteur, ce compteur n'est plus décrémenté. Des objets atteignant un tel nombre de références sont supposés rares et ne seront jamais considérés comme des *garbages* par ce reference counting, mais ils peuvent toujours être réclamés en exécutant occasionnellement une *tracing garbage collection*.

3.3.3 *One-bit reference counting*

Poussé à l'extrême, le *limited-field reference counting* utilise un seul bit contenu dans un objet pour savoir si l'objet a soit une, soit plusieurs références. Si la référence vers un objet avec une référence est supprimée, alors l'objet est lui aussi supprimé. Par contre, si l'objet avait plusieurs références, alors, la suppression d'une référence ne change rien. Cet objet ne sera donc jamais supprimé par le *one-bit reference counting*, mais une *tracing garbage collection* pourrait être effectuée périodiquement gérer ce problème. Enfin, si la taille des champs des pointeurs est trop grande pour l'espace d'adressage et que tous les *bits* dans les pointeurs ne sont donc pas utilisés, on peut en profiter pour stocker le bit supplémentaire au sein de la ou des références qui pointent vers un objet plutôt que dans l'objet lui-même, ce qui permet de ne pas utiliser de place en mémoire pour les compteurs de références. L'intérêt du *one-bit reference counting* vient du fait qu'en pratique, la plupart des objets ne sont référencés qu'une seule fois.

3.3.4 *Weighted reference counting*

La *garbage collection* distribuée dans différents espaces mémoire, voire des machines différentes est difficile à réaliser à cause des coûts du à la synchronisation et la communication entre les processus. Ils sont très élevés pour réaliser une *tracing garbage collection* qui doit pouvoir scanner l'ensemble des espaces mémoires. Une technique de *garbage collection* distribuée moins coûteuse est le *weighted reference counting*. A la place d'avoir seulement un compteur de références dans l'objet, chaque référence contient en plus un poids. Quand un objet est créé, son compteur est mis à un certain poids initial et le compteur de la première référence pointant vers cet objet est initialisé au même poids. A chaque fois qu'une référence est copiée, son poids est divisé en deux et réparti entre les deux références, c'est pourquoi le poids initial d'un objet est souvent une puissance de deux pour faciliter les divisions. Cette opération ne nécessite donc pas l'accès au compteur de l'objet (sauf si, après de trop nombreuses divisions successives, le poids d'une référence n'est plus divisible par deux). Lorsqu'une référence est supprimée, alors il faut accéder à l'objet pour soustraire le poids de la référence du compteur de l'objet. Quand ce compteur tombe à zéro, l'objet peut être réclamé.

Le *reference counting* tire donc son avantage dans le fait de détecter rapidement les *garbages* avec un mécanisme plus simple que la *tracing garbage collection*.

Il a cependant l'inconvénient majeur de ne pas suffire à lui tout seul et de nécessiter une *tracing garbage collection* à certains moments, que ce soit parce qu'il ne détecte pas les cycles ou à cause de la taille limitée du compteur de références. De plus, les accès au compteurs impliquent une charge de travail non négligeable.

3.4 Métrique de mesures des performances d'un *garbage collector*

Plusieurs métriques sont utilisées pour évaluer les performances du *garbage collector*. En voici certaines qui nous sont montrées dans les documents [3] et [13]:

- Le débit (*throughput*): il s'agit du débit de l'application. Il mesure le pourcentage de temps d'exécution qui est consacré à l'exécution du programme lui-même et non à la *garbage collection* sur des périodes des temps relativement longues.
- La surcharge de travail du *garbage collector* (*garbage collector overhead*) est le pourcentage de temps d'exécution utilisé par le *garbage collector*. Cette métrique mesure donc la même chose que le débit, mais du point de vue opposé.
- Les temps de pause (*pause times*) mesure la durée des pauses de l'application dues au *garbage collector*.
- La prévisibilité des pause (*pause predictability*): les pauses peuvent-elles être planifiées à des moments commodes pour le *mutator* plutôt que pour le *garbage collector*?
- La fréquence des collections (*frequency of collections*) par rapport à l'exécution de l'application.
- L'empreinte (*footprint*) mesure la taille qu'occupe le *heap* en mémoire.
- La rapidité de réaction (*promptness*) est la vitesse à laquelle la mémoire d'un objet devient disponible à partir du moment où cet objet est devenu un *garbage*.
- L'interaction avec la mémoire virtuelle (*virtual memory interaction*): sur des systèmes avec une mémoire physique limitée, le *garbage collector* doit éviter d'accéder à trop de pages mémoire qui ne résident pas en mémoire car le rapatriement de telles pages depuis la mémoire virtuelle vers la mémoire physique est coûteux. Il est donc intéressant que le *garbage collector* gère bien la localité des références (rapprocher des objets qui se référencent souvent de façon à ce qu'ils soient chargés ensemble lors du chargement d'une page mémoire qui peut contenir plusieurs objets). Les *compacting* ou *copying collectors* peuvent en effet profiter de la relocalisation des objets pour améliorer la localité des références.
- L'interaction avec les mémoires cache (*cache interaction*): les *garbage collections* vont souvent avoir comme effet d'enlever les données utilisées par le *mutator* qui se trouvaient dans des mémoires cache, diminuant ainsi les performances du *mutator*.
- L'impact sur le compilateur et sur l'exécution (*compiler and runtime impact*). Certains *garbage collectors* nécessitent la coopération du compilateur et de l'environnement d'exécution, comme par exemple la mise à jour de compteurs de références. Le compilateur doit en effet générer des instructions supplémentaires pour le comptage tandis que l'environnement d'exécution doit les exécuter. Quel en est l'impact sur les performances? Cela n'interfère-t-il pas avec des optimisations faites au moment de la compilation?

Une application interactive demandera plutôt des temps de pause courts, tandis qu'une non interactive favorisera plutôt le débit. Une application en temps réel peut nécessiter des garanties limitant les temps de pause et la surcharge de

travail du *garbage collector* sur une période donnée. Enfin, une petite empreinte est en général le principal souci lorsqu'on parle d'une application s'exécutant sur un PDA ou dans des systèmes embarqués.

Vers le temps réel

4 Introduction

Cette seconde partie du document nous donne tout d'abord un aperçu de ce qu'est la programmation concurrente et son utilité dans le cadre de la réalisation de systèmes en temps réel.

Elle nous explique ensuite brièvement pourquoi utiliser des *garbage collectors* lors de la réalisation de systèmes en temps réel et les problèmes que cela implique.

Enfin, des améliorations pouvant être apportées aux systèmes de *garbage collection* classiques pour palier à ces problèmes sont exposées.

4.1 Programmation concurrente

Comme nous l'explique l'ouvrage [1], la "programmation concurrente" est la partie de la programmation qui exprime le parallélisme au sein d'une application, ainsi que la communication et la synchronisation entre les entités parallèles.

4.2 Pourquoi des programmes concurrents?

Il existe trois motivations principales à écrire des programmes concurrents. La première est d'essayer de profiter au mieux du processeur. En effet, de nos jours, les opérations d'Input/Output prennent un temps considérable par rapport au temps de calcul des processeurs. Dès lors, un programme séquentiel faisant beaucoup d'I/O n'utilisera que très peu le processeur. Ensuite, écrire un programme en tenant compte de la possibilité de parallélisme peut permettre à celui-ci de profiter d'un système qui contient plusieurs processeurs. Enfin, cela nous permet de rapprocher le modèle de programmation du monde réel. En effet, pour interagir avec le monde réel, il faut pouvoir réagir en "temps réel", ce qui est rendu possible grâce à la programmation concurrente.

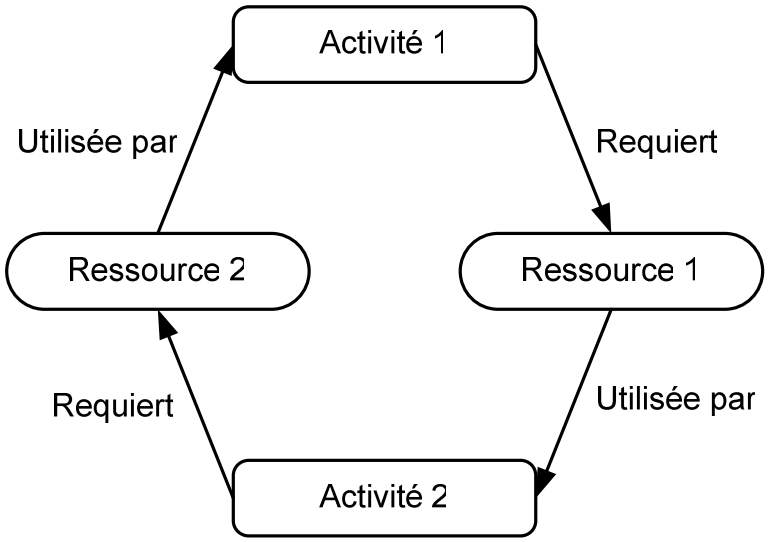
4.3 Critiques et problèmes

La principale critique apportée à la programmation en temps réel est qu'elle introduit certaines surcharges de travail pour la gestion du parallélisme et rends donc l'exécution plus lente. Cependant, les problèmes qui sont résolus par la programmation concurrente surpassent cette critique. La programmation concurrente permet en effet de se rapprocher du monde réel en exécutant plusieurs tâches "en même temps", offrant ainsi la possibilité au système de réagir à plusieurs évènements dont le déclenchement et la nature peuvent être relativement aléatoires. Gérer de telles réactions de manière aussi efficace en utilisant seulement la programmation séquentielle serait extrêmement plus complexe, voire même impossible.

Le fait d'écrire des programmes concurrents fait apparaître des problèmes qui n'existent pas dans la programmation séquentielle: les activités concurrentes doivent coordonner leurs actions. Si cette coordination n'est pas réalisée correctement, on peut avoir à faire à des problèmes qui n'existent pas dans un traitement séquentiel: *deadlock* (les activités se bloquent mutuellement), *interference* (les activités se créent mutuellement des problèmes dans leurs travaux ou se ralentissent), ou encore *starvation* (une activité ne parvient pas à progresser dans son travail).

On peut résumer le comportement que l'on désire obtenir d'un programme en deux propriétés. La première est appelée *safety* et signifie que rien de mal ne va se passer, que les différentes activités parallèles ne vont pas se causer de problèmes entre elles. La seconde, appelée *liveness*, signifie que quelque chose de bien va se passer, à savoir que toutes les activités parallèles vont progresser sans subir de *deadlock* ou de *starvation*.

Des problèmes cités ci-dessus, le *deadlock* est certainement le plus connu et le plus courant. Il se produit si les quatre conditions suivantes se produisent: exclusion mutuelle lors de l'accès aux ressources (seule une activité à la fois peut accéder à une ressource), une activité doit attendre lorsqu'elle a besoin d'une ressource déjà utilisée, une ressource ne peut être libérée que par l'activité qui l'utilise et enfin, une chaîne cyclique d'activité attendant des ressources et d'activité demandant des ressources en alternance existe.



Deadlock

Il existe trois moyens de gérer les *deadlock*:

- *Deadlock prevention*: éviter que les quatre conditions puissent être réunies en s'assurant qu'au moins une de ces quatre conditions ne se produise jamais.

- *Deadlock avoidance*: la réalisation d'aucune des conditions n'est évitée, mais les *deadlocks* sont évités à l'aide d'un algorithme les empêchant de se produire. Pour ce faire, l'algorithme analyse dynamiquement les ressources et refuse l'allocation d'une ressource qui amènerait à un *deadlock* (éviter la chaîne cyclique mentionnée ci-dessus).

- *Deadlock detection and recovery*: le problème est corrigé seulement si on entre dans l'état de *deadlock* (par exemple en terminant prématurément une des activités concurrentes qui détenait des ressources).

4.4 Systèmes en temps réel

La principale application de la programmation concurrente est la réalisation de *Real-time systems*. Ces systèmes ont pour caractéristique de devoir répondre à des stimuli d'entrée (et il s'agit même parfois du passage du temps) dans un intervalle de temps fini et spécifié. Ces systèmes sont souvent intégrés à des systèmes de production ou de contrôle industriel, par exemple. Ils sont donc intrinsèquement concurrents car ils doivent à la fois gérer les différentes données qui leur parviennent en entrée n'importe quand, mais aussi donner des sorties à ces entrée dans des temps limités, de façon à bien réaliser le contrôle en temps réel.

En plus d'être en temps réel, ces systèmes possèdent les caractéristiques suivantes:

- ils peuvent être larges et complexes, c'est-à-dire aller jusqu'à des systèmes multi-plateforme et mutli-langage de millions de lignes de codes
- il se peut qu'ils doivent être extrêmement fiables et sûrs. Il suffit de penser par exemple à la médecine ou aux usines nucléaires.
- ils doivent permettre de gérer les temps de réponse. Il est très difficile de créer un système qui assure que la réponse appropriée sera toujours donnée dans les temps. Les *real-time control facilities* permettent de spécifier quand un travail doit commencer et quand il doit se terminer, mais aussi répondre à des situations où ces timings ne sont pas respectés, mais aussi où ces timings changent dynamiquement.
- ils doivent interagir avec des interfaces matérielles. De nombreux périphériques variés peuvent à tout moment envoyer des signaux de contrôle ou d'erreur au processeur qui doit les gérer en un temps fini.
- enfin, ils requièrent une implémentation efficace et un environnement d'exécution prévisible. Même si les langages de programmation de haut niveau offrent des facilités aux programmeurs au dépend des détails de l'implémentation, les systèmes en temps réel ne peuvent se permettre de manquer de réactivité. Ils doivent donc être constamment concernés par le coût en temps des opérations ainsi que par la prédictibilité.

Il existe une distinction à faire entre les *hard real-time systems* et les *soft real-time systems*.

Les premiers sont ceux dont il est impératif que certaines tâches soient faites avant un certain temps limite, comme par exemple, les systèmes de contrôle d'un avion. Dépasser le temps pourrait amener à des catastrophes.

Dans les seconds, le temps a son importance mais le système pourra continuer à fonctionner si un temps limite est occasionnellement dépassé, comme un éditeur de texte ou une application multimédia, par exemple.

Bien entendu, beaucoup de systèmes vont être composés tant de *hard* que de *soft* sous-systèmes.

5 Garbage collection et temps réel

"La gestion statique de la mémoire était une technologie intéressante pour des raisons de sécurité et de fiabilité. Cependant, avec l'augmentation de la complexité des systèmes en temps réel, la gestion dynamique de la mémoire gérée manuellement devenait plus intéressante, mais elle amenait aussi des problèmes de prédictibilité, de fiabilité et de maintenance. Une bonne partie de ces problèmes peuvent être gérés grâce à la *garbage collection*" [16].

Même si la *garbage collection* rend le système plus fiable, elle apporte cependant de nouveaux problèmes de performance et de prédictibilité. La *garbage collection* simple interfère avec l'exécution du programme à des moments imprévisibles et pour un temps indéterminé. Cela peut empêcher la bonne exécution d'un programme en temps réel et l'empêcher des respecter des *deadlines* qui devraient être garanties.

5.1 *Generational Garbage Collection*: une augmentation du débit

Les explications suivantes sont en partie inspirées du document [5].

Comme expliqué plus tôt dans ce document, les *generational garbage collectors* divisent la mémoire en plusieurs zones appelées générations. Cela a pour but d'exploiter les observations connues sous le nom de "*weak generational hypothesis*" qui ont été faites dans plusieurs langages de programmation (incluant le Java):

- La plupart des objets ne sont pas référencés pour longtemps, c'est-à-dire qu'ils meurent jeunes
- Il y a peu de références allant des plus vieux objets vers les plus jeunes

La plupart des nouveaux objets alloués ne survivent donc pas au premier cycle de *garbage collection* après leur création, tandis que ceux qui y survivent, survivront normalement à plusieurs cycles.

Or, dans un *copying garbage collector* classique par exemple, ces objets à la vie plus longue sont non seulement scannés lors de chaque cycle de *collection*, mais ils sont en plus déplacés. Cela représente une quantité relativement importante de travail inutile.

Les *generational garbage collectors* évitent en grande partie ce travail inutile en divisant la mémoire en zones appelées des générations. Chaque objet est créé dans la génération la plus jeune et est déplacé au cours de sa vie dans les différentes générations suivant son âge.

Faire un cycle de collection dans la génération la plus jeune seulement est plus rapide que de scanner toute la mémoire. Après quoi, comme il reste en général peu d'objets dans cette génération, ces derniers sont rapidement déplacés. Par conséquent, plus une génération est jeune, plus elle sera scannée fréquemment.

De plus, différents algorithmes peuvent êtres utilisés pour effectuer la *garbage collection* dans les différentes générations, chaque algorithme étant optimisé par rapport aux caractéristiques observées en général pour une génération en particulier.

De bons choix quant à ces algorithmes, à la taille des générations, au temps que les objets doivent passer dans chaque génération, ainsi qu'à la façon de disposer les différentes générations en mémoire permettent d'optimiser au mieux le travail.

Dans sa forme la plus simple et la plus utilisée, le *generational garbage collector* utilise une zone de la mémoire appelée "*nursery*" pour la création des nouveaux objets. Celle-ci est scannée plus souvent que la mémoire en entier et les objets qui survivent à un cycle de *collection* dans cette zone sont déplacés vers un autre endroit en mémoire. On peut donc considérer qu'il y a deux générations.

En résumé, les nouveaux objets créés sont placés dans la génération la plus jeune et quand l'espace alloué à cette génération est plein, le *garbage collector* utilise les pointeurs racine et les pointeurs inter-générations pour collecter seulement les *garbages* de la génération la plus jeune, sans toucher aux autres.

Les objets qui survivent à un certain nombre de collections sont copiés dans la génération suivante et quand une génération devient pleine, celle-ci et toutes les plus jeunes sont collectées.

5.1.1 *Inter-generational pointers*

Les *tracing garbage collectors* commencent tous à scanner la mémoire depuis le *root set* et traversent les références entres les objets jusqu'à ce que tous les objets atteignables aient été visités.

Les *garbage collectors* générationnels commencent eux aussi par le *root set*, mais ne suivent pas les références qui mènent dans les générations plus vieilles que la génération qui est scannée, ce qui réduit la quantité d'objets à scanner, mais crée un problème. Si un objet se trouvant dans la génération scannée est référencé par un objet d'une génération plus vieille et qu'aucune chaîne de référence partant du *root set* ne permet de l'atteindre, cet objet sera considéré comme *garbage* alors qu'il ne l'est pas.

C'est la raison pour laquelle les *garbage collector* générationnels doivent suivre la trace des pointeurs allant d'un objet dans une vieille génération vers un objet dans une plus jeune. Des pointeurs de ce type sont appelés "pointeurs inter-générations" ("*intergenerational pointers*"). Ils peuvent être créés soit en modifiant une référence dans un "vieil" objet pour référencer un "jeune" objet, soit lorsqu'un objet qui est promu vers une plus vieille génération référence d'autres objets plus jeunes.

5.1.2 *Tracking intergenerational references*

Un premier moyen de trouver les pointeurs inter-générations est de scanner linéairement les générations plus vieilles à leur recherche. Cette opération n'est pas des moins coûteuses mais elle reste tout de même plus rentable que d'exécuter l'algorithme de *tracing* à travers toutes les générations.

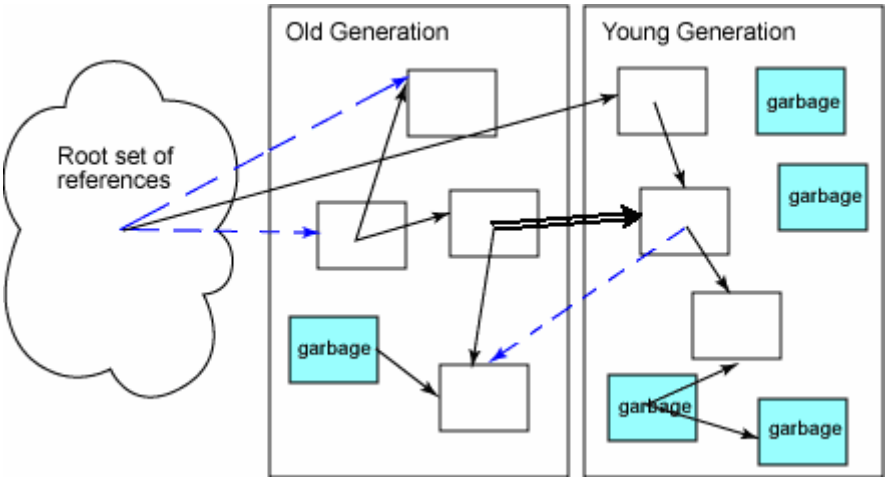
L'autre moyen d'opérer est de garder en mémoire l'ensemble de ces références et de maintenir cet ensemble à jour. Le *garbage collector* peut simplement lui-même mettre l'ensemble à jour lorsqu'il déplace un objet dans la génération suivante.

Il reste donc à mettre l'ensemble à jour lorsqu'une référence inter-générations est créée lors de la modification d'un pointeur par le *mutator*. Il existe différentes techniques pour y parvenir.

Une première manière de procéder est de faire générer au compilateur des instructions englobant l'assignation d'un pointeur. Ces instructions pourraient au besoin mettre à jour l'ensemble des références inter-générations comme le fait le *reference counting* lors de la mise à jour d'un compteur. Le *garbage collector* pourrait aussi utiliser un système de protection virtuel pour attraper les accès aux vieilles générations. Enfin, une autre approche potentiellement plus efficace est d'utiliser le *bit* de modification d'une page en mémoire pour identifier des blocs modifiés que l'on pourrait alors scanner linéairement.

Il est possible d'éviter l'inspection du pointeur à chaque modification de référence. Il n'est par exemple pas nécessaire de prendre en comptes les enregistrements de pointeurs dans des variables locales ou statiques, car elles font déjà partie du *root set*. Les enregistrements de pointeurs dus à l'initialisation d'un nouvel objet n'ont pas besoin d'être pris en compte non plus, car (presque) tous les objets sont alloués dans la plus jeune génération.

En tous les cas, le *garbage collector* doit connaître l'ensemble des pointeurs inter-générations lors de la collection d'une jeune génération et ajouter ceux-ci au *root set* pour éviter de "faux *garbages*".



Pointeurs inter-générations
"Intergenerational references" provenant du document [5]

Dans ce schéma où les flèches représentent des références entre des objets, la double-flèche au milieu du schéma est une *intergenerational reference*. Cette référence doit être ajoutée au *root set* pour une collection de la jeune génération seulement. Par contre, les flèches pointillées sont les références qui ne doivent pas être suivie lors d'une telle collection.

5.1.3 *Card marking*

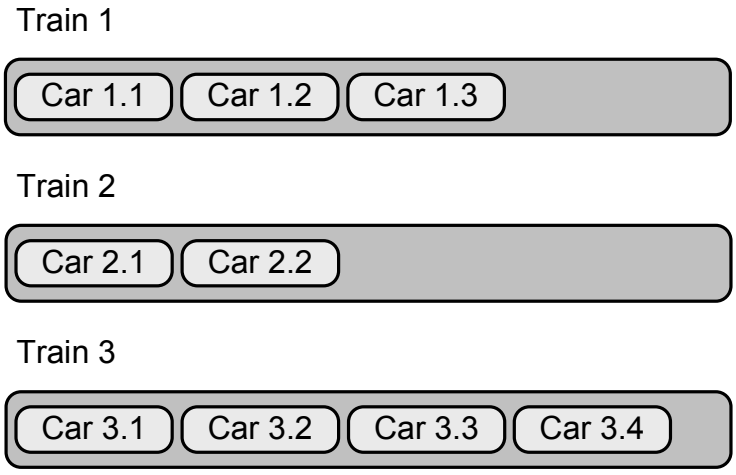
Le *card marking* est une technique qui permet de repérer des modifications de pointeurs contenus dans les objets de vieilles générations. Avec cette technique, le *heap* est divisé en un ensemble de "cartes" (*cards*), habituellement plus petites qu'une page mémoire. Une table faisant correspondre un *bit* (ou un *byte*, suivant l'implémentation) à chaque *card* est gardée en mémoire. Lorsqu'un pointeur dans un objet est modifié, le *bit* dans la table correspondant à la *card* où se trouve l'objet est activé. Lors de la *garbage collection*, les *cards* correspondant aux *bits* activés sont scannées à la recherche de pointeurs inter-générations. Après cela, les *bits* sont réinitialisés.

Même s'il permet de ne pas traquer tous les pointeurs inter-générations pendant l'exécution, le *card marking* n'est pas sans coût:

- la place requise pour la table des *bits*
- travail d'activation des *bits* à chaque modification de pointeur (excepté leur initialisation)
- travail supplémentaire lors de la *garbage collection*: il faut en effet parcourir la table et scanner toutes les *cards* correspondant à un *bit* activé

5.1.4 *The train collector*

Le "*train collector*" est un procédé qui divise l'espace des objets en blocs de taille fixe. L'algorithme libère les *garbages* en s'occupant d'un bloc à la fois. Pour ce faire, il regroupe les blocs appelés "voitures" dans différents espaces disjoints appelés "trains". Les voitures appartiennent donc à un seul train et sont ordonnées au sein de celui-ci dans l'ordre de leur arrivée dans le train. Les trains sont quant à eux ordonnés dans l'ordre de leur création. Une voiture précède une autre si elle se trouve dans un train plus vieux ou si elle se trouve dans le même train et a été ajoutée à ce train plus tôt.



Disposition en mémoire pour un *train collector*
Créé à partir du schéma *"The object space for the train collector"* du document [8]

A chaque invocation du *garbage collector*, celui-ci s'occupe de la voiture la plus jeune (dans le train le plus jeune). Pour ce faire, il commence par vérifier s'il existe une référence externe vers le train dans lequel se trouve la voiture en cours de collection. Si ce n'est pas le cas, tout le train contient des *garbages* et toutes ses voitures sont réclamées. Sinon, tous les objets de la voiture en cours de collection qui sont référencés de l'extérieur sont déplacés dans le train où se trouvent les références, regroupant ainsi les objets qui se référencent pour améliorer leur localité.

Les objets référencés depuis les *root pointers*, sont évacués dans n'importe quel train autre que celui en cours de collection. Les objets ainsi évacués sont eux aussi scannés à la recherche de pointeurs dans la voiture en cours de collection, auquel cas les objets référencés sont évacués à leur tour, etc. Quand il ne reste plus de référence extérieure vers la voiture collectée, elle ne contient plus que des *garbages* et l'espace mémoire qu'elle occupait peut être récupéré pour terminer la collection.

Pour chaque voiture, on retient l'ensemble des références qui pointent vers elle. Cet ensemble est maintenu à jour grâce à des *write barriers* (cfr point 5.2.4). Comme les voitures sont collectées de la plus jeune à la plus vieille, l'ensemble des références extérieures peut être amélioré en ne gardant que les références venant

des voitures plus vieilles. De cette façon, lorsqu'on libère l'espace occupé par une voiture, il n'est pas nécessaire de purger les ensembles des références extérieures des voitures plus vielles.

La taille d'une voiture fait normalement 2^n bytes. Cela permet de maintenir une table des trains à laquelle on accède rapidement en faisant un *shift* de l'adresse de **n** bytes vers la droite et en utilisant le résultat comme index.

5.1.5 Discussion

Les *real-time garbage collectors* doivent fournir certaines garanties valables dans le pire des cas pouvant se produire. Les *garbage collectors* générationnels vont cependant privilégier les performances moyennes au détriment de ces garanties. Ils seront donc bons pour certaines applications de *soft real-time* comme par exemples des applications multimédia.

Les *garbage collectors* générationnels présentent trois améliorations importantes:

- les performances moyennes
- la localité des références
- le temps de latence: plusieurs petites pauses plutôt que peu de grandes pauses

Mais ils présentent aussi des désavantages majeurs:

- ils ajoutent une charge de travail
- le coût d'enregistrement des pointeurs inter-générations est proportionnel à la vitesse d'exécution du programme plutôt qu'à la vitesse d'allocation. En effet, plusieurs instructions doivent être exécutées pour chaque mise à jour de pointeur. Or, celles-ci se produisent tout au long de l'exécution pas seulement lors des allocations. Ce coût est parfois un des coûts principaux de ces *garbages collectors*.

Pour des applications ayant des contraintes strictes de temps réel, une *garbage collection* incrémentale en finesse est nécessaire. En effet, pour ce type d'applications, le *garbage collector* ne peut pas se permettre de bloquer le programme et de faire tout un cycle en une seule action atomique. Il faut donc découper le travail du *garbage collector* en petites unités qui pourront être exécutées de façon entrelacée avec l'exécution du programme.

5.2 *Incremental garbage collection*: une diminution des temps de pause

Comme expliqué dans le document [16], ce type de *garbage collection* est la première technique qui vise à implémenter un *garbage collector* non intrusif. Le travail du *garbage collector* est ici divisé en incréments qui peuvent être entrelacés avec l'exécution de l'application. Un incrément est l'exécution d'une (petite) partie du travail d'un cycle de collection. Cette quantité représente un certain nombre d'instructions ou une certaine quantité de mémoire collectée, ou encore un certain temps.

On se rend compte que rendre incrémental un *reference-counting garbage collector* n'est pas difficile, étant donné que la détection des *garbages* peut se faire facilement au cours de l'exécution du programme quand un compteur arrive à zéro. Cependant, c'est déconseillé pour des raisons d'efficacité et de performances.

En ce qui concerne les *tracing garbage collectors*, rendre incrémentale la détection des *garbages* n'est pas une chose facile. En effet, pendant que le *garbage collector* trace le graphe des objets atteignables, le programme peut modifier ce graphe pendant son exécution parallèle. Il faudra donc aussi suivre les changements qui sont faits au graphe.

5.2.1 *Incremental Copying Scheme*

Le document [8] nous explique que, comme dans sa version non incrémentale expliquée auparavant, l'*incremental copying scheme* sépare la mémoire en deux espaces: "*from-space*" et "*to-space*". La première étape de la collection consiste à copier tous les objets atteignables depuis les pointeurs racine dans le *to-space*.

Au cours de l'exécution entrelacée, tout objet se trouvant dans le *from-space* auquel le programme accède est copié dans le *to-space* grâce à un système de "*read barrier*" (cfr. point suivant). Les nouveaux objets alloués sont quant à eux alloués dans le *to-space*.

A la fin de la collection, les objets restants dans le *from-space* sont réclamés et les noms des deux espaces sont inversés.

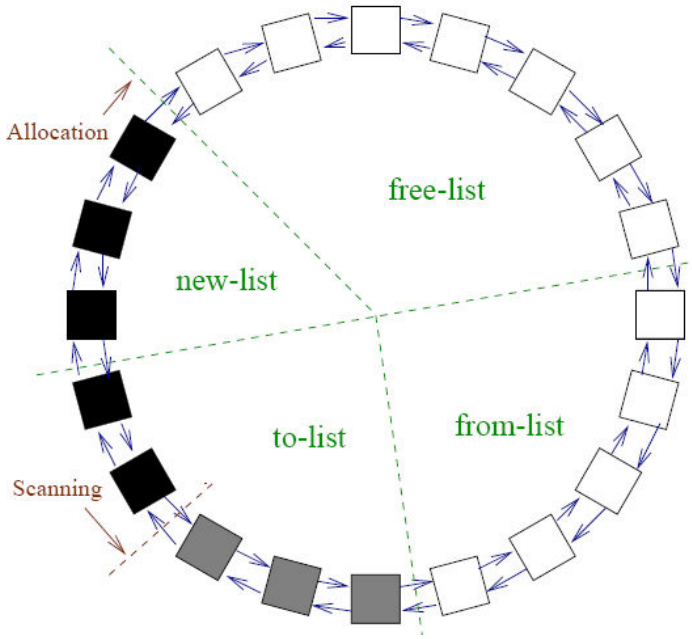
5.2.2 *Read Barriers*

La *read barrier* (barrière de lecture) est un système qui consiste à exécuter un certains nombre d'instructions lors de toute lecture de certains pointeurs. Elles peuvent par exemple être utilisées par les *garbage collectors* pour gérer l'exécution entrelacée du *mutator* et du *garbage collector*. Au cours d'un cycle de collection, la *read barrier* permet d'éviter les interférences entre les deux exécutions en exécutant des instructions mettant à jour l'état du *garbage collector* lorsque le *mutator* accède aux objets.

Les *read barriers* peuvent être implémentée grâce à des instructions software précédant la lecture potentielle de pointeurs du *heap*. Cependant, les implémenter de cette façon est relativement coûteux en travail (Sur une machine conventionnelle, le surcoût du à l'opération est de plus ou moins dix pourcents). Cet algorithme nécessite en effet que tous les pointeurs racine soient scannés en une seule opération atomique. Comme alternative, les *read barriers* peuvent être implémentées de manière hardware ou grâce à de petites routines. Une autre approche utilise le système de mémoire virtuelle pour que les vérifications des *read barriers* soient réalisées par l'unité matérielle de gestion mémoire: MMU (*Memory Management Unit*). Malheureusement, avec cette approche, le coût lors du pire scénario est trop élevé pour être accepté par des tâches en temps réel strict.

5.2.3 *Incremental Implicit Reclamation*

Un autre *collector* incrémental est l'algorithme de réclamation implicite de Baker[1] dont le document [8] nous donne un aperçu, qui est une *non-copying* version du procédé incrémental. Ce *garbage collector* ajoute deux pointeurs et une couleur (blanc, gris ou noir) comme donnée à chaque objet. Les pointeurs servent à lier les objets entre eux dans une liste doublement chaînée cyclique. Cette technique, appelée *treadmill* (manège) divise cette liste en quatre sous-listes. On trouve tout d'abord la *from-list* et la *to-list* qui correspondent respectivement au *from-space* et au *to-space*. Ensuite vient la *new-list*. Celle-ci contient les nouveaux objets, qui sont alloué en noir. Enfin vient la *free-list*, qui contient une liste d'emplacements vides en mémoire. La *from-list* contient les objets alloués avant que la *garbage collection* ne commence et qui doivent donc encore être scannés par le *garbage collector*. Donc, pendant l'exécution de celui-ci en parallèle avec celle du programme, lors passage du *garbage collector*, tous les objets atteignables dans la *from-list* sont peu à peu déplacés vers la *to-list* qui est initialement vide. La mémoire occupée par les *garbages* est quant à elle libérée à la fin du cycle de *garbage collection*.



Les quatre listes du *treadmill garbage collector*
"*A representation of the four lists of a treadmill GC*" provenant du document [8]

[1] H.G.Baker, "The Treadmill: Real-Time Garbage Collection without Motion Sickness", in *Proc. Of the Workshop on Garbage Collection in Object-Oriented Systems.* OOPSLA'91, 1991.

Segregation

Pour être sur que l'exécution du *garbage collector* se termine avant qu'il n'y ait plus de mémoire libre, le travail des incréments doit être évalué d'après le rythme des allocations. Si tous les objets étaient de même tailles, les allocations seraient facilement exécutées à un coût constant. Cependant, c'est rarement le cas en pratique et un emplacement libre suffisamment grand doit être recherché dans une liste de zones libres lors d'une allocation. Le fait de gérer des objets de différentes tailles entraîne non seulement un problème de fragmentation, mais aussi des temps d'allocation non déterministes. Une variante de l'algorithme expliqué ci-dessus utilise plusieurs structures de *treadmill* qui sont chacune liées à la gestion d'une catégorie d'objets, les objets étant classés dans les différentes catégories selon leur taille. Cette technique est appelée *segregation*. Pour éviter un trop grand nombre de catégories de taille, la taille des objets est arrondie à la plus proche puissance de deux supérieure. Ceci permet de résoudre le problème des temps d'allocation non déterministes.

5.2.4 *Write Barriers*

La *write barrier* (barrière d'écriture) est une technique similaire aux *read barriers* à la différence que les *write barriers* exécutent des instructions supplémentaires lors de la modification de certains pointeurs, c'est-à-dire lors de l'écriture de certains pointeurs et non de leur lecture.

La technique des *write barriers* empêche l'application de créer un pointeur allant d'un objet noir (déjà scanné par le *garbage collector*) vers un objet blanc (*garbage* potentiel), évitant ainsi de le non respect de l'invariant des trois couleurs cité au point 3.1.1 (aucun objet noir ne pointe vers un objet blanc). Ceci est nécessaire pour être certain que le *garbage collector* ne libèrera pas la mémoire d'un objet atteignable.

Deux conditions sont requises pour que le *garbage collector* manque un objet atteignable:

- l'application créée un référence vers un objet blanc dans un objet noir
- tous les chemins allant d'un objet gris (qui doit encore être scanné) vers cet objet blanc sont détruits

L'objet qui n'est pourtant pas un *garbage* (étant donné qu'il est référencé par un objet noir) sera considéré comme tel étant donné que le *garbage collector* a déjà scanné et ne scannera plus l'objet qui contient la référence et qu'il ne trouvera aucune autre référence vers cet objet dans les objets qu'il doit encore scanner (les gris).

Il y a deux stratégies de bases pour implémenter des *write barriers*: *incremental update* et *snapshot-at-beginning*.

Les algorithmes du premier type s'assurent que la première condition ne peut être réalisée en colorant l'objet noir ou l'objet blanc. Ils sont qualifiés d'*incremental update* (mise à jour incrémentale) parce qu'ils mettent à jour la vue du graphe des objets du *garbage collector* en fonction des changements fait par l'application.

Les algorithmes du second type vont quant à eux s'assurer que la deuxième condition ne va pas se produire. Pour ce faire, ils vont forcer le *garbage collector* à

traiter toute référence que l'application écrase et qui pourrait faire partie d'un chemin mentionné dans la condition. Leur nom provient du fait qu'ils gardent la trace des références qui existaient au début du cycle de collection.

Le document [19] expose une implémentation de *write barrier* qui réduit la surcharge de travail dans le *mutator* à seulement deux instructions supplémentaires par *store* vérifié.

5.3 Gestion mémoire par régions

La gestion mémoire par régions ("*Region-based Memory Management*") est un compromis qui ne nécessite pas de *garbage collection*, mais ne requiert pas non plus du programmeur qu'il se charge de chacune des libérations de mémoire.

Avec ce système de gestion mémoire, les allocations peuvent se faire dans différentes régions en mémoire.

L'allocation et la libération des régions elles-mêmes peuvent par exemple être déterminées à la compilation par une analyse basée sur les types. Dans d'autres cas, les allocations et libérations de régions sont explicites dans le langage.

L'utilisation de régions a l'avantage d'être prévisible et performante. De plus, une bonne utilisation permet d'augmenter la localité des références.

Enfin, dans la RTSJ expliqué plus loin dans ce document, l'allocation d'une région est explicite mais sa libération ne peut être faite tant qu'elle est utilisée par une *thread*. La libération des régions mémoire est effectuée automatiquement grâce à un compteur de références pour chaque région.

6 Autres améliorations potentielles de la *garbage collection*

Ce qui suit présente certaines idées tirées du document [16] car celui offre une façon originale de voir les choses au sein d'une solution relativement complète étudiée de la théorie jusqu'à des tests.

Il nous propose en effet de nouvelles idées pour améliorer la *garbage collection* en se concentrant sur le *scheduling* du *garbage collector* plutôt que sur l'algorithme lui-même. Ce document explique qu'il est possible de calculer une *deadline* à laquelle la *garbage collection* doit être terminée pour les nouvelles allocations mémoire. Avec cette *deadline* explicite, il est possible de planifier les *garbage collector* en utilisant des techniques de *scheduling* standard, comme par exemple le "*earliest deadline first scheduling*" (*EDF Scheduling*) qui consiste à planifier la tâche dont la *deadline* est la plus proche en premier. Comme le temps détermine quand déclencher le *garbage collector*, cette approche est appelée *time-triggered garbage collection*.

Ce document se concentre aussi sur le problème de la gestion du non-déterminisme pour les applications en temps réel. Une approche réussie dans ce domaine est le *feedback scheduling*. En utilisant le *feedback control*, le *scheduling* s'adapte dynamiquement pour garder le taux d'utilisation du processeur à un niveau correct. Ce document s'intéresse donc à une gestion de la mémoire adaptive dans le but de la rendre plus robuste dans un environnement changeant.

Le document présente aussi une nouvelle approche d'application de priorités pour l'allocation de mémoire et montre comment cela peut augmenter la robustesse et les performances des systèmes en temps réel.

Enfin, il propose aussi une vérification expérimentale des techniques proposées. La suite est un aperçu de ce que nous explique ce document.

6.1 *Incremental garbage collection* et temps réel

De façon à garantir l'avancement du travail du *garbage collector*, on peut décider d'exécuter un certain nombre d'incréments à chaque demande d'allocation de mémoire. Le document [16] nous cite l'algorithme de Baker énoncé auparavant comme exemple d'algorithme incrémental. Cependant il faut décider de la quantité de travail qui sera faite en fonction de chaque allocation.

Pour ce faire, posons:

- F_{min}: la quantité minimale de mémoire disponible pour l'allocation pendant un cycle de *garbage collection*.
- a: la quantité de mémoire requise
- W_{max}: la quantité maximale de travail de *garbage collection* qui peut être requise pour compléter son cycle (selon une certaine métrique et une unité correspondante).

Alors, la quantité de travail (w) qui doit être exécutée en fonction de l'allocation pour garantir que le système ne tombe pas à cours de mémoire avant que le cycle de collection soit terminé est :

$$w \geq W_{max} \cdot \frac{a}{F_{min}}$$

Exemple: supposons qu'il reste au moins 1000 Ko de mémoire libre (F_{min}), que le cycle de *garbage collection* en cours soit terminé dans au maximum 10000 instructions (W_{max}) et qu'une allocation de 50 Ko de mémoire survienne (a). Alors, le nombre d'instructions (w) qui doivent être exécutée pour cette allocation est d'au moins

10000 instructions * 200 Ko / 1000 Ko = 2000 instructions.

Ainsi, quand la quantité de mémoire disponible diminue de un cinquième, le nombre d'instructions de *garbage collection* restantes avant la fin du cycle diminue aussi de un cinquième. De cette façon, le cycle sera terminé et donc les *garbages* collectés avant qu'il n'y ait plus de mémoire libre.

Ce genre de *garbage collectors* incrémentaux déclenchés par des allocations a cependant deux désavantages majeurs. Tout d'abord, même si la surcharge de travail qu'il entraîne est moins importante, des nombreuses allocations les unes à la suite des autres peuvent entraîner des délais. De plus, de façon à garder le coût en travail de chaque incrément en dessous d'une certaine limite, il faut utiliser des métriques complexes pour pouvoir décider quand il faut arrêter chaque incrément. Une métrique trop simple apporte en effet une pauvre approximation du comportement du *garbage collector* dans le temps.

Prenons l'exemple d'une métrique basée sur le nombre d'objets collectés. Un incrément court selon cette métrique peut en réalité prendre beaucoup de temps. Le problème vient du fait qu'il puisse falloir scanner une quantité importante de pointeur avant de trouver un seul objet que l'on peut libérer. On ne peut donc pas garantir une limite au temps nécessaire pour trouver un objet.

Exécuter la *garbage collection* au moment de l'allocation permet de prouver facilement que le *garbage collector* arrivera à suivre la demande de l'application. Cependant, cette technique a comme inconvénient d'exécuter à chaque fois le *garbage collector* pendant l'exécution des *threads* du programme – causant donc des interférences – plutôt que de profiter de moments où le processeur est moins occupé.

Une façon de surmonter ce problème est de rendre le travail du *garbage collector* concurrent, c'est-à-dire, assigner le travail du *garbage collector* à une *thread* s'exécutant en parallèle avec celles de l'application. Cette stratégie a été appliquée pour certains *garbage collectors*, mais elle n'a pas été très utilisée dans le contexte du temps réel. En effet, rien ne garanti que le *garbage collector* va suivre la demande d'allocation de l'application.

De façon à satisfaire les demandes des *hard real-time systems*, on doit trouver une technique pour agencer le travail d'un *garbage collector* concurrent de façon à garantir à l'application d'arriver à temps à toutes ses *hard deadlines*.

6.2 Réponse à nombre limité de *threads* en temps réel

Considérons un système classique qui comporte un nombre limité de *threads* de haute priorité, c'est-à-dire qu'elles ont certains délais à respecter strictement (périodiquement en général). Dans les systèmes embarqués, il existe en général un nombre limité de telles *threads*. Les autres *threads* de priorité plus basse sont souvent exécutées avec des délais plus souples.

Partant de ce point de vue, on peut concevoir un système où le travail de *garbage collection* n'est jamais effectué lorsqu'une *thread* de haute priorité est exécutée. A la place, on assignera le travail de *garbage collection* engendré par ces *threads* de haute priorité à une autre *thread* de *garbage collection* qui est exécutée quand aucune des *threads* de haute priorité n'est en cours d'exécution. Cette *thread* devra donc effectuer une quantité de travail proportionnelle à la quantité de mémoire allouée par les *threads* de haute priorité.

Notons que ce *garbage collector* peut temporairement prendre un peu de retard dans son travail, comme il n'est pas exécuté immédiatement pour les *threads* prioritaires. Il doit donc toujours y avoir une certaine quantité de mémoire réservée pour ces *threads*.

Certaines analyses permettent de calculer la quantité de mémoire à réserver, mais aussi de vérifier que le *garbage collector* va toujours suffire pour suivre la demande des *threads* prioritaires.

Quant au travail de *garbage collection* engendré par les *threads* de basse priorité, il est effectué incrémentalement en correspondance avec les allocations, comme dans le principe expliqué ci-dessus.

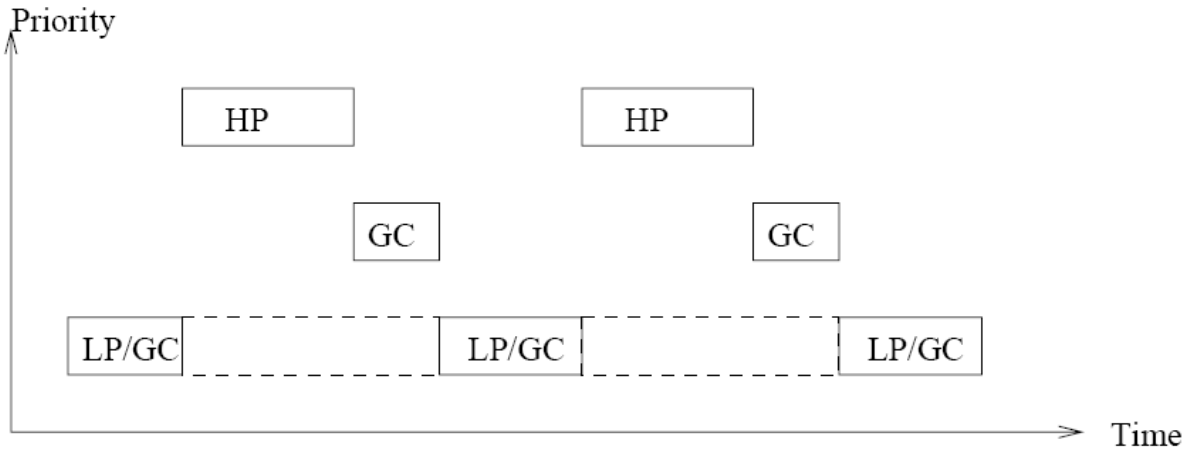
Comme le travail de *garbage collection* est effectué en partie en concurrence et en partie incrémentalement, cette approche est appelée *semi-concurrent scheduling*.

On trouve trois niveaux de priorités dans un système utilisant une telle stratégie d'agencement:

-. Les *threads* de haute priorité

-. Celle de *garbage collection* requise pour satisfaire aux *threads* de haute priorité.

-. Et enfin, les *threads* de basse priorité et leur *garbage collection* incrémentale.



Exemple d'agencement avec ses trois niveaux de priorité
HP (*high priority*): *threads* de haute priorité
GC (*garbage collector*): *thread* de *garbage collection* pour les objets des *threads* de haute priorité
LP/GC (*low priority / garbage collector*): *threads* de basse priorité et *garbage collector* pour leurs objets

Cette technique a donc pour avantage de permettre de garantir certaines performances de temps réel strict aux *threads* qui le nécessitent. De plus, comme le travail de *garbage collection* des *threads* de basse priorité n'est pas effectué pendant l'exécution de celles de haute priorité, ce *garbage collector* va pouvoir utiliser une métrique plus simple sans affecter les performances des *threads* de haute priorité.

Cette approche a quand même certains désavantages. L'un d'eux est qu'elle ne convient pas directement aux systèmes avec des "*EDF schedulers*". Un autre inconvénient est qu'il faut toujours faire une quantité importante d'analyse de *scheduling* pour adapter le *collector* à une plateforme spécifique.

6.3 *Time-triggered garbage collection*

Les *incremental garbage collector*s traditionnels planifient leur travail en fonction des allocations de l'application: pour chaque unité d'allocation, une certaine quantité de travail est réalisée par le *garbage collector*. Une approche différente consiste à utiliser le temps plutôt que les allocations comme déclencheur du travail de collection. De ce point de vue, la *garbage collection* est planifiée pour terminer à un moment donné plutôt qu'après une quantité de travail déterminée.

Les principaux domaines où cette *time-triggered garbage collection* a un impact sont:

Un *garbage collector* concurrent dans un système basé sur des *deadlines*: pour pouvoir organiser la *garbage collection* de façon à pouvoir donner des garanties de temps réel tout en perturbant le moins possible les autres *threads* de

l'application, il faut pouvoir planifier le *garbage collector* de la même manière que toute autre *thread*.

Les métriques de travail du *garbage collector*: un *garbage collector* incrémental traditionnel se fie à certaines métriques pour déterminer s'il est en phase avec l'application ou s'il doit réaliser plus de travail. Le *garbage collector* se fie donc à cette métrique et ne pas utiliser une bonne métrique peut mener à de mauvaise performance pour le temps réel. Les erreurs causées par une moins bonne métrique peuvent être évitées en utilisant la métrique optimale de travail du *garbage collector*: le temps CPU réel qui est requis pour arriver à la fin d'un cycle de collection.

Allocations en rafales: les applications allouent souvent la mémoire par rafales. Ce qui signifie que les exécutions d'un *garbage collector* déclenchées par les allocations se feraient aussi par rafales. Un *time-triggered garbage collector* n'est pas concerné par ce problème étant donné qu'un cycle de collection doit se terminer avant un certain temps limite, sans prêter attention au moment où l'application réalise des allocations.

Unified garbage collection scheduling: les *schedulers* ("planificateurs") de *garbage collection* basés sur une métrique traditionnelle sont étroitement liés à l'implémentation du *garbage collector*. En utilisant une approche basée sur le temps pour la planification du travail, il devrait être possible de séparer le *scheduler* de l'algorithme de *garbage collection*: en effet, utiliser le temps comme déclencheur et comme métrique fournit une interface simple entre le *garbage collector* et le *scheduler*.

Le document [16] nous explique aussi comment il est possible en théorie de calculer une limite maximale au temps nécessaire pour achever le cycle de *garbage collection* qui garantisse de ne pas tomber à cours de mémoire, c'est-à-dire une *deadline* pour ce cycle. Il nous explique ensuite comment bien utiliser le temps comme métrique de travail du *garbage collector*.

Enfin, il parle de la façon d'implémenter une *time-triggered garbage collector* dans un système à priorités fixes et basé sur les *deadlines* et la manière donc le processus général de *scheduling* affecter le *scheduling* de la *garbage collection*.

6.4 *Adaptive Garbage Collection Scheduling*

Pour pouvoir donner des garanties par rapport aux deadlines en temps réel, on procède à des analyses en considérant le pire des cas (*worst-case analysis*). Ces analyses sont relativement difficiles, et ce d'autant plus pour un *garbage collector* dans un système *mutlithreaded* car le travail du *garbage collector* ne dépend pas seulement de l'application mais aussi de la planification des *threads*. De plus, on se base sur les performances de la mémoire, ce qui est relativement non-déterministe sur des mémoires avec des caches, etc. Enfin, les analyses dans le pire des cas sont les plus pessimistes, ce qui peut amener à une faible utilisation du processeur.

Pour éviter les problèmes inhérents aux analyses dans le pire des cas, on peut faire s'adapter le *scheduler* aux différentes exigences de l'application grâce à un système de "*feedback control*".

Le *scheduling* des *threads* et du *garbage collector* s'adapterait en fonctions de la situation.

Pour ce faire, le *garbage collector* peut être rendu *auto-tuning*, c'est-à-dire qu'il va lui-même choisir certains de ses paramètres en fonction de la situation.

Le document [16] présente aussi des techniques pour estimer automatiquement en pratique la longueur et la quantité de travail nécessaire à un cycle de *garbage collection*.

Il fournit une nouvelle méthode d'application de priorités aux allocations. Les mécanismes proposés peuvent aussi être utilisés pour améliorer les performances de système avec gestion automatique de mémoire en limitant la quantité de travail du *garbage collector*. Une façon d'introduire des priorités pour l'allocation mémoire dans un système Java sans devoir changer la syntaxe du langage est aussi proposée. Elle a été implémentée dans une machine virtuelle Java expérimentale et vérifiée dans une application de contrôle automatique.

Enfin, le document propose un support expérimental aux techniques proposées. La plateforme de test est un système de contrôle simple qui déplace une balle sur une barre. L'angle de cette dernière est utilisé pour déplacer la balle à un endroit voulu. Le contrôle a été effectué par une application Java composée de trois *threads*.

En Java

7 Introduction

Le Java est un jeune langage de programmation conçu en 1991 et qui a principalement attiré l'attention de la communauté des programmeurs en 1994 en s'orientant vers Internet [1]. C'est en effet à partir de ce moment que sa popularité a grandit en s'appuyant sur la vague provoquée par Internet.

Il offrait en effet certains avantages sur ses concurrents: orienté objet et familier, robuste, distribué et sécurisé, portable, performant, dynamique et enfin, *multithreaded*.

Le langage semblait être la base idéale lors de l'apparition des systèmes en temps réel. Cependant, il se révéla qu'il avait de sérieuses limitations pour la programmation en temps réel, ce qui a amené au développement de la *Real-Time Specification for Java* (RTSJ). Il s'agit d'une extension du langage basée sur des librairies de classes qui facilitent la programmation en temps réel, souvent en améliorant le modèle de *Threading* du langage de base.

8 Programmation concurrente en Java

Nous avons montré au point 4.2 l'utilité et la nécessité d'utiliser des modèles de programmation concurrents. Les mécanismes permettant de réaliser de la programmation concurrente peuvent être fournis de deux façons différentes: soit par un système d'exploitation, soit par un langage lui-même. Les langages C et C++ sont des langages séquentiels et ils ne supportent donc pas explicitement la programmation concurrente. Ils doivent donc faire appel à des *Applications Programmers' Interface* (API) pour utiliser les possibilités offertes par le système d'exploitation pour y parvenir. Par contre, des langages de programmation comme le Java (ou encore le C#) sont des langages de programmation dits "concurrents", c'est-à-dire qu'ils supportent explicitement ce type de programmation. Dans ces deux langages, une activité concurrente est appelée une *thread*.

8.1 *Process, thread,* et *fiber*

Le terme *process* (processus) représentait à ses débuts une séquence d'actions réalisées par l'exécution d'une séquences d'instructions. Un processus concurrent est donc une activité séquentielle qui peut donc être réalisée en même temps que d'autres processus.

Notons que la justesse d'un programme concurrent (composé de plusieurs processus parallèles) ne doit pas dépendre de l'ordre d'exécution des sous-processus. Si c'est le cas, les contraintes de synchronisation des sous-processus doivent être programmées explicitement.

Les systèmes d'exploitation fournissent des possibilités de création de processus concurrents qui s'exécutent indépendamment les uns des autres, dans leur propre "machine virtuelle", évitant ainsi les interférences.

Par après, des processus pouvant être créés au sein d'un langage sont apparus. Ceux-ci sont appelé *threads* (ou parfois *tasks)* et ont un accès sans restriction à une mémoire partagée. On parle donc ici de concurrence au sein d'un programme et non de concurrence entre programmes, comme pour les processus.

Enfin, des *fibers* sont des *threads* qui sont invisibles depuis le système d'exploitation. Celles-ci sont gérées par des libraires de routines au niveau applicatif.

En Java, les *threads* sont gérées par la *Java Virtual Machine* (JVM). On distingue donc les *native threads*, visibles par le système d'exploitation, des *green threads* qui sont implémentées directement par la JVM.

8.2 Communication et synchronisation

Peu importe leur nom ou représentation, des activités concurrentes peuvent avoir besoin de communiquer et se synchroniser pour bien pouvoir coopérer. On peut grossièrement classer les moyens d'y parvenir en deux catégories: les variables partagées et le passage de messages.

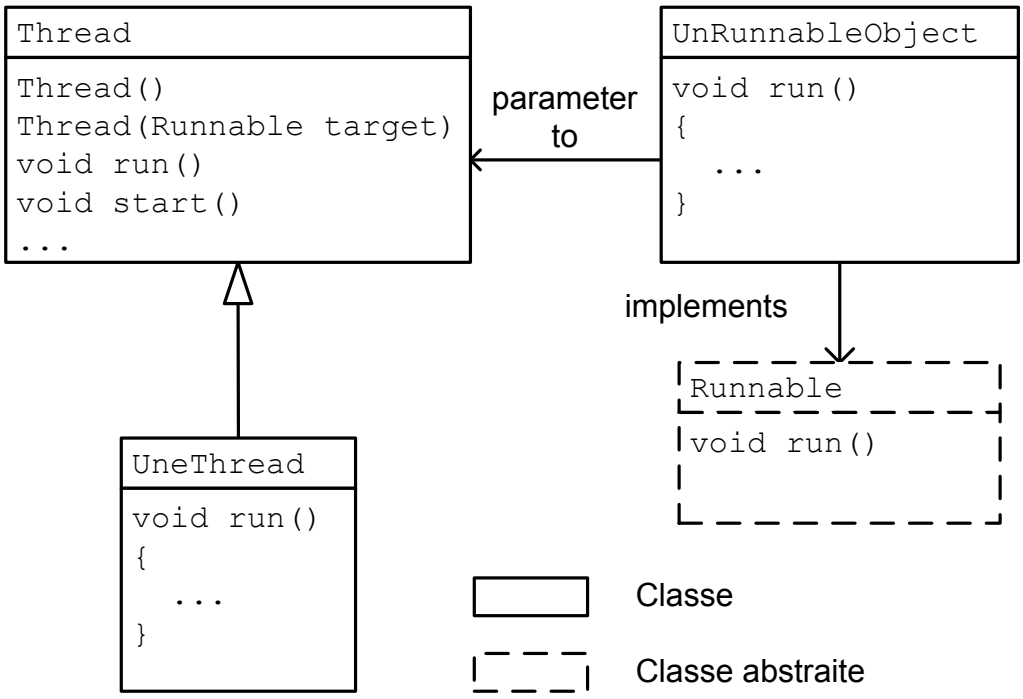
L'approche qui a servi de base pour la communication et la synchronisation en Java est une approche assez populaire appelée *monitor*. Un *monitor* encapsule une ressource et fournit une interface pour l'exécution d'opérations en exclusion mutuelle vers cette ressource. Un *monitor* n'est donc possédé que par une seule *thread* à la fois. Cela est généralement implémenté à l'aide d'un *lock*.

8.3 Aperçu du modèle de concurrence du Java

Comme le Java était un nouveau langage, il a pu être directement implémenté en intégrant la concurrence au sein du modèle sans se soucier de la compatibilité avec des versions précédentes, comme c'est le cas pour certains langages auxquels cette notion a été ajoutée par après. Le modèle de concurrence du Java utilise le concept d'objet actif (*active object*): chaque *thread* sera créée en instanciant un objet de type `Thread` qui possède dès lors les méthodes `run` et `start`.

Pour démarrer une *thread*, il faut donc appeler la méthode `start` sur l'objet correspondant. Cette méthode démarre la nouvelle *thread* qui exécutera le contenu de la méthode `run`. Dès lors, une application peut définir le travail d'une *thread* en redéfinissant la méthode `run` dans une sous-classe de `Thread`. Un second moyen de définir le travail d'une *thread* est de passer un objet implémentant l'interface `Runnable` (méthode `run`) en argument lors de la création d'une `Thread`.

Notons que si l'on appelle directement la méthode `run` d'une `Thread` et non la méthode `start`, la *thread* n'est pas démarrée et le travail ne sera pas exécuté en parallèle, mais bien en séquentiel (par la *thread appelante*) comme une invocation classique de méthode. C'est donc bien l'appel à la méthode `start` qui lance un nouveau travail en concurrence.



Deux façons d'utiliser les `Thread`: définir une sous-classe de `Thread` redéfinissant la méthode `run` (`UneThread`) ou définir une classe implémentant l'interface `Runnable` (`UnRunnableObject`) et passer une instance de cette classe comme paramètre au constructeur de la classe `Thread`.
Sur base du schéma *"Thread Creation"* du document [1]

Enfin, la communication entre *threads* est réalisée via des objets Java. Tout objet en Java possède au besoin un *monitor* et donc un *lock* d'exclusion mutuelle, protégeant ainsi l'objet de mises à jours simultanées pouvant provoquer des interférences. De plus, une méthode en Java peut posséder le libellé `synchronized`. Une *thread* ne peut exécuter une méthode *synchronized* d'un objet qu'après avoir acquis le *lock* de son *monitor*.

8.4 Fin d'une *thread*

La fin normale d'une *thread* se produit lorsque celle-ci termine l'exécution de la méthode `run`. Cependant, cette fin peut aussi être provoquée par d'autres moyens. La *thread* s'arrête aussi si une exception est lancée dans le corps de la méthode `run` sans être attrapée. Elle s'arrête aussi si on n'appelle la méthode `destroy` ou `stop` sur l'objet correspondant à la *thread*. La différence entre ces deux méthodes est que la méthode `stop` lance une exception qui peut permettre à la *thread* de s'achever plus proprement alors que la méthode `destroy` ne permet pas à la *thread* de "nettoyer" avant de s'arrêter. Dans les deux cas, il ne s'agit pas d'une fin normale et ces deux méthodes ont d'ailleurs été *deprecated* par après.

Notons aussi qu'il existe deux types de *threads*: les *user threads* et les *daemon threads*. C'est le premier type qui est utilisé par défaut, mais l'appel de la méthode `setDaemon(true)` sur un objet `Thread` (avant le `start`) permet d'en

faire une *daemon thread*. Normalement, ce type de *threads* fournit un service et ne s'arrête pas de toute l'exécution du programme, elle s'arrête une fois que toutes les *user threads* se sont terminées.

Enfin, toutes les *threads* d'un programme se terminent lors de l'appel à la méthode

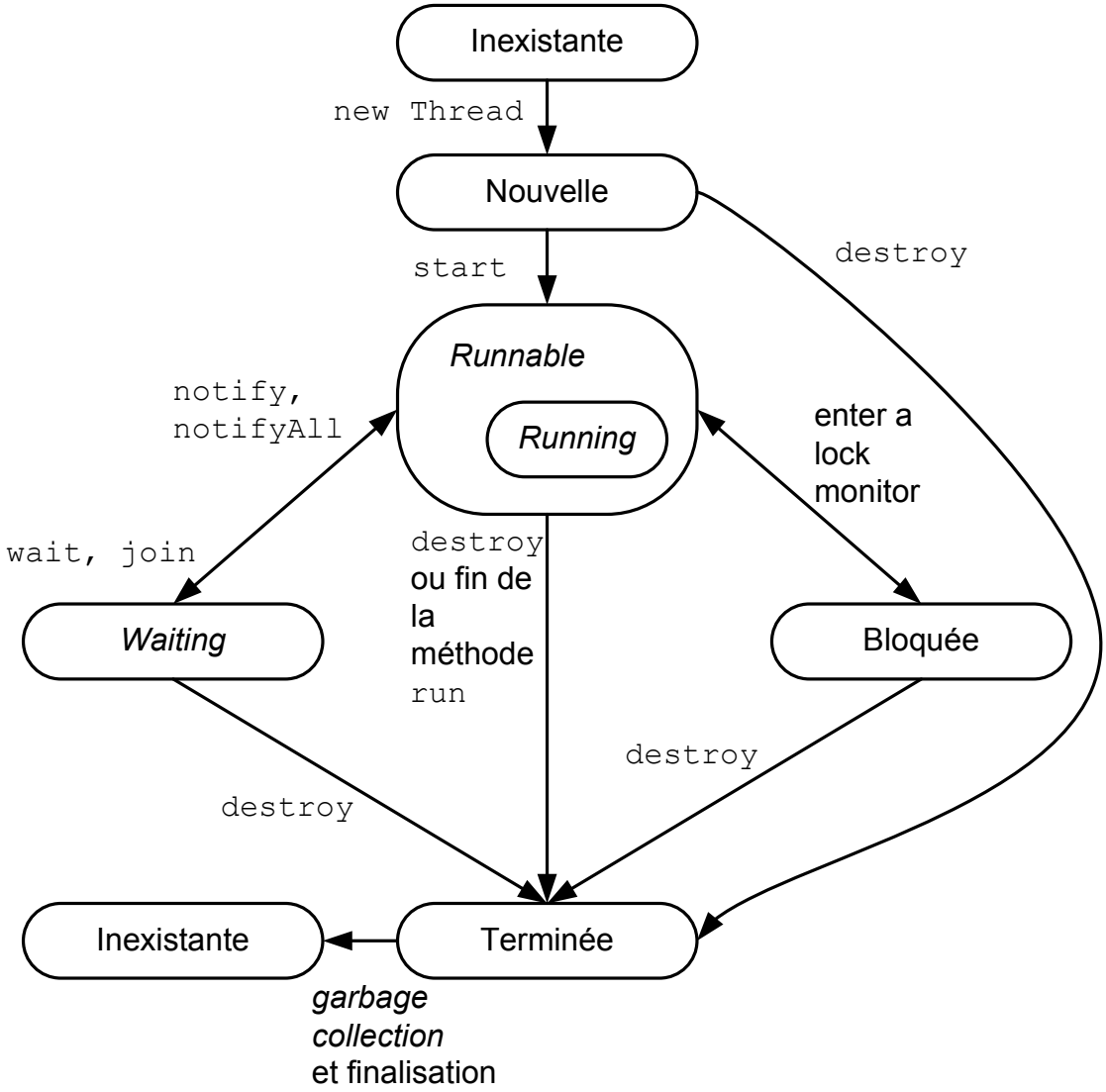
```
System.exit(int status)
```

Notons enfin que l'on peut spécifier certaines *threads* qui seront chargée de la bonne terminaison du programme lors de l'appel à cette méthode `exit` grâce à la méthode

```
System.addShutdownHook(Thread hook)
```

Les *threads* qui auront été passées en argument à cette méthode seront démarrées lors de l'appel à la méthode `exit`. On peut cependant arrêter le programme sans démarrer ces *threads* en faisant appel à la méthode `halt` de la classe `System`.

8.5 Cycle de vie d'une *thread*



Cycle de vie d'une *thread*
Sur base du schéma *"Simple State Transition Diagram for a Thread"* [1]

On peut voir sur ce schéma que toute *thread* doit d'abord être créée en appelant le constructeur `new Thread` et qu'à partir de ce moment, elle peut passer à l'état terminé en appelant la méthode `destroy` et ce, peut importe l'état dans lequel elle se trouvait. Notons aussi qu'une des *threads* dans l'état *runnable* peut avoir la main sur le processeur et se trouver donc plutôt dans l'état *running*. Enfin, les méthodes `wait`, `join`, `notify` et `notifyAll` vont être expliquées au point 8.7: communication et synchronisation.

8.6 Données locales à une *thread*.

Comme nous l'avons dit plus haut, les données en mémoire sont partagées par les différentes *threads*, les attributs sont donc partagés entre les différentes *threads*, qu'ils soient des attributs de classe ou non. Cependant, il est possible de créer des attributs qui pourront prendre une valeur différente pour chaque *thread*. Pour ce faire, il suffit que ceux-ci soient de type `ThreadLocal`.

8.7 Communication et synchronisation

8.7.1 *Locks* et mot-clé *synchronized*

Comme nous l'avons vu, un *lock* d'exclusion mutuelle est associé à chaque objet. Celui-ci ne peut être directement accédé par une application, mais est affecté par le mot-clé `synchronized` et les *synchronization blocks* (bloc de synchronisation).

Une méthode d'un objet libellée par le mot-clé `synchronized` ne peut être accédée par une *thread* qu'après l'obtention du *lock* associé à l'objet. Une méthode sans ce mot-clé peut quant à elle être accédée à tout moment sans avoir besoin d'obtenir le *lock*. Dès lors, pour assurer une exclusion mutuelle complète lors de l'accès à des données encapsulées, toutes les méthodes accédant à ces données doivent être libellées `synchronized`.

Les blocs de synchronisation, quant à eux, prennent la forme suivante:

```
synchronized (objet) { /* code */ }
```

Le mot-clé `synchronized` prend en paramètre un objet dont le lock doit être obtenu avant de pouvoir exécuter le code contenu entre accolades.

On peut donc implémenter le mot `synchronized` associé à une méthode à l'aide d'un bloc de synchronisation de la façon suivante:

```
methode() { synchronized(this) { /* corps méthode */} }
```

Notons enfin que ces blocs permettent d'exprimer des contraintes de synchronisation plus avancées, mais qu'à cause d'eux, on ne peut pas avoir une vue d'ensemble des synchronisations associées à un objet en regardant simplement le code de la classe, comme c'était le cas avec les méthodes synchronisées.

Il n'est en général pas possible de déterminer le groupe de *lock* que détient une *thread*, mais la classe `Thread` fournit cependant une méthode permettant de dire si la *thread* courante détient le lock d'un objet ou non:

```
public static boolean holdsLock(Object obj)
```

De plus, elle fournit aussi une méthode (et une énumération) permettant de connaître l'état d'exécution d'une *thread* et donc de savoir si celle-ci attends l'obtention d'un *lock* (`BLOCKED`).

```
public static final enum State {BLOCKED, NEW, RUNNABLE,
TERMINATED, TIMED_WAIT, WAITING};
```

et

```
public State getState();
```

En Java, pour chaque classe, un objet de type `Class` est associé. Dès lors, en ce qui concerne la synchronisation par rapport aux variables statiques, elle revient à obtenir le *lock* de l'objet `Class` correspondant à la classe de la variable. Cela peut être réalisé soit en libellant une méthode statique avec le mot-clé `synchronized`, soit en créant un bloc synchronisé et en lui passant l'objet `Class` comme argument.

8.7.2 *Waiting* et *Notifying*

Les méthodes `wait`, `notify` et `notifyAll` sont toutes trois définies dans la classe `Object` et sont donc héritées par tout objet en Java. Ces méthodes ont été définies pour être utilisées depuis le corps de méthodes qui requièrent que la *thread* appelante ait obtenu le *lock* de l'objet (et qui sont donc `synchronized`), sans quoi une `IllegalMonitorStateException` est levée.

La méthode `wait` bloque la *thread* appelante et libère le *lock* associé à l'objet que la *thread* avait obtenu. Cette méthode peut prendre un argument spécifiant un nombre de millisecondes pendant lesquelles la *thread* doit attendre pour lui éviter d'attendre indéfiniment. Elle peut aussi accepter un second argument permettant d'amener à la précision de la nanoseconde quand cela est possible. Donc, même sans être notifiée (voir paragraphe suivant), la *thread* sera donc réveillée un fois le temps écoulé (ou un peu après). Enfin, appelée sans arguments, aucun *timeout* n'est associé au `wait` et la *thread* peut attendre indéfiniment (cet appel est équivalent à ceux des méthodes avec paramètres avec comme arguments des zéros).

La méthode `notify` permet quant à elle de débloquer une *thread* qui est en attente après un appel à la méthode `wait`. Celle-ci est choisie au hasard parmi les *threads* qui sont endormies sur l'objet. Enfin, la méthode `notifyAll` réveille toutes les *threads* endormies sur l'objet.

Notons qu'une *thread* réveillée par un `notify` ou `notifyAll` doit préalablement réobtenir le *lock* de l'objet avant de pouvoir continuer.

Utilisation

Comme nous l'avons dit ci-dessus, la *thread* qui sera réveillée par un notify ne peut être choisie. C'est pourquoi une *thread* qui se réveille doit en général vérifier que c'est bien à son tour de se réveiller, elle doit donc réévaluer sa condition

à chaque réveil et éventuellement se rendormir. Le `wait` s'utilise donc en général selon la forme suivante:

```
while(condition) wait();
```

Dans le cas du `notifyAll`, il est évident que les *threads* doivent réévaluer leur condition.

Depuis la version 1.5 du Java, quelques utilitaires relatifs à la concurrence ont été ajoutés dans le package `java.util.concurrent`. Citons par exemple le package `java.util.concurrent.locks` qui contient l'interface `Lock` et une implémentation: `ReentrantLock`. On y trouve des méthodes telles que `lock` et `unlock` et permet de créer des *locks* et d'y associer des `Condition`. L'interface `Condition` contenant entre autres les méthodes `await` et `signal` appartient aussi à ce package. `ReentrantLock` en contient une implémentation dans une classe interne.

8.7.3 Contrôle asynchrone des *threads*

Des versions plus anciennes du Java proposaient des méthodes telles que `stop`, `suspend` et `resume` qui permettaient à une *thread* d'avoir un certain contrôle sur une autre. Ces méthodes sont maintenant `deprectated` car leur utilisation est malsaine (peut mener à des *race conditions* ou des *deadlock*) et elles devraient donc être évitées.

Il reste cependant une autre forme plus limitée de contrôle asynchrone via la méthode `interrupt`. Appeler cette méthode sur une `Thread` revient à activer un flag. Cependant, il appartient à cette `Thread` de tester ce flag via les méthodes `isInterrupted` et `interrupted` (cette dernière est une méthode de classe qui teste si le flag de la *thread* courante a été activé et réinitialise ce flag). Enfin, pour qu'une *thread* puisse appeler la méthode `interrupt` sur une autre, elle doit en avoir les permissions nécessaires et la *thread* cible doit être dans l'état runnable (et non bloquée par un `sleep`, un `wait` ou un `join`), sinon une exception sera levée.

8.7.4 Retarder une *thread*

Une *thread* peut se différer en faisant appel à la méthode statique `sleep` de la classe `Thread`. Cette méthode permet donc à une *thread* d'attendre un temps futur sans devoir constamment lire l'horloge système. Le temps à attendre peut être spécifié en milliseconde, voir en nanosecondes (bien que peu de systèmes puissent le supporter).

Il est important de comprendre que la *thread* n'aura pas le processeur dès que le temps demandé sera écoulé. Tout d'abord, il peut exister des différences de granularité entre le temps spécifié et l'horloge système. Ensuite, l'implémentation du `sleep` peut avoir recours à un `interrupt` système. Or, il peut arriver que ceux-ci soient désactivés pendant de courtes périodes de temps. Enfin, la *thread* sera remise dans l'état *runnable*, mais devra attendre son tour avant de passer à l'état

running et ainsi avoir le processeur pour travailler. Le temps spécifié comme argument de la méthode `sleep` est donc un temps minimum. On ne peut donc pas utiliser cette méthode pour assurer qu'un travail sera exécuté toutes les 50 millisecondes, par exemple.

Notons enfin que le temps spécifié en argument est un temps relatif. Le langage ne fournit en effet pas le moyen de spécifier un temps absolu, c'est-à-dire qu'une *thread* doit se réveiller à tel moment tel jour.

Il en est de même lorsqu'un timeout est spécifié lors de l'utilisation de la méthode `wait`, le temps d'attente maximal est spécifié de façon relative. De plus, il est impossible de savoir après cette méthode si la *thread* a été réveillée à cause du timeout ou par un `notify`, le `wait` n'ayant pas d'argument de retour.

8.7.5 *Thread Groups* (Groupes de *threads*)

Les `ThreadGroup` permettent à un ensemble de *threads* d'être groupées et manipulées en tant que groupe plutôt qu'individuellement. Les groupes de *threads* sont organisés de façon hiérarchisée, c'est-à-dire qu'un groupe de *threads* peut contenir des *threads*, mais aussi être le parent d'autres groupes de *threads*. Toute *thread* ou groupe de *threads* qui est créé appartient à un *thread group*. Il existe un groupe par défaut auquel est associé la *thread* `main`. Toute *thread* ou *thread group* qui est crée sans spécifier un *thread group* est placé dans le même *thread group* que celui associé à la *thread* créatrice. Enfin, chaque *thread group* possède une priorité maximum qu'aucune *thread* du groupe ne peut dépasser, sans quoi celle-ci est tronquée.

8.8 Priorité des *threads* et *scheduling*

Une priorité peut être associée à une *thread* par le programmeur via la méthode `setPriority` de la classe `Thread`. La priorité d'une *thread* peut être obtenue grâce à la méthode `getPriority`. Cette priorité est un entier situé entre `Thread.MIN_PRIORITY` (=1) et `Thread.MAX_PRIORITY` (=10) et elle est assignée par défaut à la valeur de la *thread* parent. La valeur par défaut de la priorité de la *thread* exécutant la méthode `main` est `Thread.NORM_PRIORITY` (=5).

Du point de vue du temps réel, ces contraintes de priorité sont faibles. En effet, il n'y a aucune garantie que la *thread* de plus haute priorité est tout le temps en cours d'exécution. De plus, le Java ne garantit pas que les exécutions des *threads* de même priorité seront bien entrelacées pour assurer une répartition correcte du processeur. Enfin, quand des *threads* natives sont utilisées, des *threads* de priorité différente peuvent être ramenées à la même priorité du point de vue du système d'exploitation.

8.9 Utilitaires relatifs à la concurrence

Un utilitaire du Java permet pouvoir programmer des tâches à exécuter grâce aux classes `TimerTask` et `Timer` du package `java.util`. Une tâche à programmer doit implémenter la méthode `run` de la classe abstraite `TimerTask`. Ensuite, un `Timer` doit être créé et les tâches à planifier peuvent lui être transmises pour qu'il se charge de les exécuter. Notons aussi qu'une tâche peut être planifiée de façon répétée.

Jusqu'à présent, nous avons considéré la concurrence au sein de la JVM. Cependant, le langage Java fournit aussi des moyens d'interagir avec d'autres processus du système exploitation, externes à la JVM. Ces moyens sont fournis au via la classe abstraite `java.lang.Process` et la classe `java.lang.Runtime`. Un processus est représenté par un objet héritant de la classe `Process`. Le seul moyen d'obtenir un tel objet est via l'objet de la classe `Runtime`. Cette classe utilise le pattern du singleton et on ne peut donc en obtenir la seule instance que via la méthode statique `Runtime.getRuntime`. Une fois cette instance obtenue, on peut y appliquer des méthodes telles que la méthode `exec` qui permet d'exécuter un processus et renvoie l'objet de type `Process` correspondant.

8.10 Forces et limitations du modèle de concurrence du Java

Cette partie résume la critique du modèle de concurrence du Java se trouvant dans [1].

La principale force du modèle de concurrence du Java réside dans le fait qu'il est simple et directement supporté par le langage, sans avoir recours aux primitives de concurrence du système d'exploitation. La syntaxe et le typage fort du langage permettent aussi d'éviter certaines erreurs de programmation.

Cependant, le langage est peut-être un peu trop simple et n'est donc pas suffisamment expressif pour de la programmation d'application complexes en temps réel. On peut résumer les lacunes du langage dans les lignes suivantes.

Il y a tout d'abord un manque en ce qui concerne la programmation à l'aide de variables de conditions. Les implémentations de ces variables avec ce que le langage nous fournit risquent trop de provoquer des problèmes comme des *deadlock* et manquent de performance.

Les possibilités de programmation en utilisant des références absolues au temps sont manquantes.

On ne peut pas non plus favoriser une *thread* réveillée par un `notify` par rapport aux autres *theads* en attente d'un moniteur.

Il y a une difficulté de repérer les appels au moniteur d'objet imbriqués (ce qui peut mener à des *deadlocks* lorsqu'une *thread* possède le *lock* de plusieurs moniteurs et doit attendre) ou de savoir si un objet est fiable par rapport à la programmation concurrente (voir l'échelle de Bloch ci-dessous).

Les moyens fournis pour donner préférence à certaines *threads* via des priorités sont insuffisants. Les *threads* Java ne sont pas suffisamment expressives en ce qui concerne la programmation en temps réel. Par exemple, la plupart des activités en temps réel ont des *deadlines*, mais il n'y a aucune notion de *deadline* au sein du langage Java.

Bloch a proposé en 1991 l'échelle suivante qui permet de caractériser des classes selon leur niveau de fiabilité par rapport à la programmation concurrente. Les niveaux de cette échelle sont les suivants:

Immutable: Les instances de cette classe sont constantes. Comme elles ne changent pas, elles ne risquent rien lors de l'utilisation avec des *threads*. Un exemple de ce type de classes est la classe `String`.

Thread-Safe: Ce type de classe peut être utilisé sans danger avec des *threads*. Les méthodes sont proprement synchronisées.

Conditionally thread-safe: Une classe contenant des méthodes qui sont déjà *thread-safe* et d'autres qui requièrent que le *lock* soit gardé entre des appels de méthodes en séquence (Pour obtenir deux valeurs contenues dans l'objet, par exemple). Cela peut être réalisé en utilisant les blocs synchronisés. Ces classes sont donc *thread-safe* à condition d'être bien utilisées.

Thread-Compatible: Les méthodes de ce type de classe ne fournissent pas de synchronisation. Cependant, il est possible de les utiliser correctement dans environnement concurrent à condition que l'appelant se charge de la synchronisation.

Thread-hostile: Les instances de ces classes ne devraient pas être utilisées dans un environnement concurrent. Idéalement, aucune classe ne devrait être écrite de cette façon. Typiquement, il s'agit de classes qui accèdent à des données statiques ou de l'environnement externe.

9 La machine virtuelle Java

La machine virtuelle Java (*Java Virtual Machine*) est au cœur des plates-formes Java et Java 2. C'est l'élément qui permet l'indépendance avec le matériel et le système d'exploitation, la taille réduite du code compilé et le moyen de protéger l'utilisateur de programmes malveillants.

La JVM est une machine de calcul abstraite. Tout comme une vraie machine de calcul, elle a un ensemble d'instructions et manipule la mémoire lors de son exécution. La JVM est décrite dans une spécification: la *Java Virtual Machine Specification* [18], permettant à ceux qui le veulent d'implémenter une machine virtuelle java. Cette spécification en est à sa seconde édition et on peut trouver les deux éditions à l'adresse http://java.sun.com/docs/books/jvms/.

Une JVM est capable d'exécuter du code compilé depuis le langage Java, qui est spécifié dans le document intitulé *The Java™ Language Specification* (http://java.sun.com/docs/books/jls/) qui en est lui à sa troisième édition. La JVM ne connaît rien du langage de programmation Java en lui-même à part un format binaire particulier: le format de fichiers "*.class*". Ces fichiers peuvent être créés en compilant du code Java et contiennent des instructions pour la JVM (appelés *bytecodes*) et quelques autres informations comme par exemple une table des symboles.

Pour des raisons de sécurité, la JVM impose des contraintes de format et de structure strictes sur ces fichiers. Cependant, n'importe quel langage dont on peut exprimer les fonctionnalités en termes de fichier .class valide peut être accueilli par la JVM.

Une implémentation de machine virtuelle Java parmi d'autres a été réalisée par la société Sun Microsystems. Il s'agit d'un des composant des produits "*Java™ 2 SDK*" (JDK) et "*Java™ 2 Runtime Environment*" (JRE) qui permettent de programmer et d'exécuter du Java et sont disponibles pour différentes plates-formes. Il s'agit sans doute d'ailleurs de la plateforme Java la plus utilisée. Cependant, la spécification de la JVM ne suppose aucune implémentation de technologie, de matériel ou encore de système d'exploitation en particulier.

La spécification de la JVM donne une vue d'ensemble de la structure de la JVM. Elle spécifie aussi entre autres l'ensemble des instructions supportées et le format des fichiers *.class*, ainsi que la façon de les exécuter.

Par contre, les détails d'implémentation ne font pas partie de la spécification pour ne pas trop limiter la créativité des implémenteurs, ce qui pourrait par exemple empêcher exemple certaines optimisations internes de la JVM.

9.1 *Java Virtual Machine Memory Management*

La spécification de la JVM définit certaines zones des données comme par exemple les registres "*pc*" (*program counter*) ou encore les *stacks* des *threads*.

La spécification définit aussi certains aspects du *heap*. Il s'agit de la zone de données dans laquelle sont alloués les instances d'une classe et les tableaux créés à l'aide du mot-clé `new`. La spécification précise que les zones mémoire occupées par les objets ne sont jamais explicitement désallouées, mais qu'elles sont récupérées par la JVM grâce à un système de gestion automatique, connu sous le nom de *garbage collector*. La JVM n'impose pas de disposition particulière des données en mémoire et ne présuppose aucun type de système de gestion de données. Les techniques de gestion de données utilisées peuvent donc être choisies en fonction des besoins du système de l'implémenteur. Le *heap* peut être de taille fixe ou non, auquel cas il peut être étendu ou compacté au besoin. La mémoire pour le *heap* peut être continue ou non.

Une implémentation de la JVM peut fournir au programmeur le contrôle sur la taille initiale du *heap*, ainsi que sur sa taille maximale et minimale, si celle-ci est dynamique.

Enfin, une condition exceptionnelle est associée au *heap*: s'il y a plus de mémoire requise qu'il n'y en a de disponible ou que le système de gestion automatique de la mémoire ne peut en libérer, la JVM lancera une *OutOfMemoryError*.

10 La plateforme Java de Sun Microsystems

Comme expliqué au point précédent, l'un des composants principaux du JDK et du JRE est l'implémentation qu'ils contiennent de la Java Virtual Machine. Le *Java Runtime Environment* est un environnement d'exécution Java qui, une fois installé sur une machine, permet principalement d'exécuter des applications en Java. Le *Java SDK*, (*Java Software Development Kit*) est, comme son nom l'indique, destiné aux personnes qui souhaitent développer des programmes en Java. Il contient donc lui aussi le nécessaire pour exécuter des programmes en Java, mais aussi de quoi créer des programmes Java, comme par exemple un compilateur. Ces deux produits peuvent être obtenus gratuitement sur site de la société Sun.

La partie qui va suivre nous explique l'évolution des versions du JDK (et du JRE) pour permettre de bien comprendre l'évolution des *garbage collectors* au sein de la JVM qui nous est expliquée après.

Ces deux produits, tout comme l'implémentation de la JVM qu'ils contiennent ont en effet connu de nombreuses versions depuis leur apparition. Les différentes versions du JRE correspondent bien entendu aux différentes versions du JDK.

Le document [22] nous donne un aperçu de ces versions au fil des années. La première version du JDK à avoir été publiée est la version 1.0 et a été mise gratuitement à la disposition du public sur internet le 23 janvier 1996.

Elle fut suivie le 19 février 1997 par la version 1.1 qui ajoutait certaines fonctionnalités, comme par exemple un dispositif robuste d'accès aux bases de données appelé *JDBC* ou encore des fonctionnalités d'accès à distance aux objets (*RMI*). Cette version apportait aussi une première modification au langage Java lui-même avec l'introduction des classes internes.

Le 8 décembre 1998, une nouvelle version sort dont le nom de code est *Playground*. Cette version de la plateforme et les suivantes ont été renommées sous le label de *Java* 2. Le JDK qui en est à sa version 1.2 est par la suite appelé "*J2SE*" 1.2 (*Java 2 Platform, Standard Edition*) pour distinguer cette plateforme de base des plateformes *J2EE* (*Java 2 Platform, Enterprise Edition*) et *J2ME* (*Java 2 Platform, Micro Edition*).

En plus des modifications d'appellation, des ajouts ont bien entendu été faits à la plateforme. Citons par exemple l'ajout de l'API graphique "*Swing*" et de quelques autres composants au noyau. Un compilateur JIT, permettant de transformer du *bytecode* en code machine ainsi que certaines autres fonctionnalités ont aussi été ajoutées.

Cette version apporte aussi des améliorations au niveau de la gestion mémoire: une allocation plus rapide et une *garbage collection* améliorée.

Enfin, l'architecture de la plateforme a aussi été modifiée de façon à pouvoir accueillir une autre implémentation de la JVM, prévoyant ainsi l'arrivée de la nouvelle version de la JVM de Sun: la *Hotspot JVM*.

La version 1.3 dont le nom de code est *Kestrel* est sortie le 8 mai 2000. La principale changement de cette version est qu'elle comprenait maintenant la *Hotspot*

JVM (celle-ci est sortie en avril 1999 et pouvait être ajoutée comme extension à la J2SE 1.2).

La version suivante est la 1.4 et est l'une des versions qui apporte le plus de changements majeurs. Elle a pour nom de code *Merlin* et est la première distribution de la plateforme à avoir été développée selon le processus "*Java Community Process*". Celui-ci à pour but de veiller à la bonne évolution de la plateforme en utilisant un processus qui permet de recevoir des requêtes quant aux nouvelles fonctionnalités à ajouter à la version suivantes de la plateforme.

Cette nouvelle version ajoute un bon nombre de nouvelles fonctionnalités. Citons entre autres *Java Web Start* permettant d'obtenir et d'exécuter simplement une application depuis un browser par exemple. Des extensions de sécurité et cryptographie ont aussi été ajoutée, ainsi que la compatibilité avec certains formats d'images comme le JPEG ou le PNG.

La J2SE 1.5 sortie le 30 décembre 2004 et a pour nom de code *Tiger*. La numérotation des versions change et cette version est maintenant appelée "J2SE 5.0".

Citons comme ajout à cette versions les "*generics*": il s'agit d'une modification au langage qui permet certaines vérifications de types supplémentaires à la compilation permettant de se passer d'un bon nombre de *cast* dans le code. Le langage supporte aussi maintenant la conversion automatique de types primitif en classes qui y correspondent (comme par exemple le `int` et la classe `Integer`). Signalons aussi l'apparition de méthodes à nombres d'arguments variables ainsi qu'une nouvelle syntaxe pour les boucles `for`.

La version actuelle, sortie le 11 décembre 2006, a apporté une nouvelle nomenclature au nom de la plateforme en remplaçant le nom J2SE par "*Java SE*" et en enlevant le ".0" du numéro de version. Cette version dont le nom de code est *Mustang* est donc appelée "Java SE 6". Des versions de tests étaient publiées pendant la phase de développement apportant petit à petit des corrections de *bugs* et des améliorations.

Cette version apporte entre autres des améliorations graphiques et de performances.

Enfin, la version suivante prévue est la Java SE 7. Son nom de code est *Dolphin* et ce projet a débuté en août 2006.

Au fil des ans, cette plateforme a subi de nombreux changements en plus de ceux apportés au langage Java lui-même. Les librairies de classes sont passées de quelques centaines de classes dans la première version à plus de 3000 classes dans la J2SE 5.0. De nouvelles API entières comme par exemple *Swing* ont été ajoutées et de nombreuses classes et méthodes de la première version sont maintenant `deprecated`.

10.1 Le *garbage collector* de la JVM de Sun Microsystems

Cette partie est inspirée d'informations collectées dans les documents [13], [17], [4], [7], [6], et [3].

Le *garbage collector* a lui aussi évolué au fil des différentes versions de la plateforme Java de Sun. Les premiers JDK utilisaient des *garbage collectors* à *thread* unique de type *mark-and-sweep* ou *mark-and-compact*.

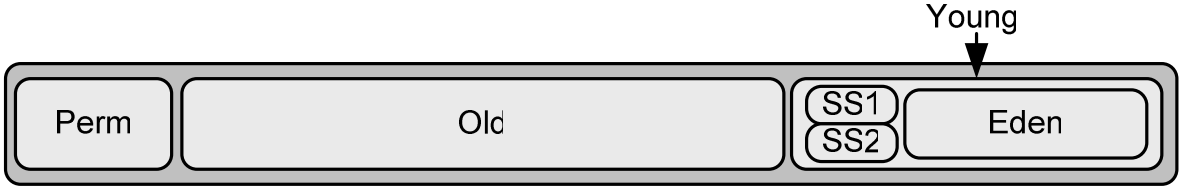
C'est à partir de la version 1.2 que le JDK a utilisé l'approche de la *generational collection*.

Cependant, cette version et les précédentes utilisaient des *conservatives* ou *partially-accurate garbage collectors*. Ce n'est qu'à partir de l'arrivée de la *Hotspot JVM* que le *garbage collector* est devenu *fully-accurate*, offrant la garantie que tous les objets inaccessibles en mémoire peuvent effectivement être réclamés. De plus, les objets pouvaient dès lors êtres délocalisés, permettant ainsi le compactage de la mémoire pour éliminer la fragmentation et augmenter la localité des objets en mémoire.

10.1.1 Principe de base

Le principe de base des *garbage collectors* du JDK depuis la version 1.2 utilise une approche mélangeant différentes techniques de *garbage collection*. Le *heap* est découpé en différentes zones. Il est composé de deux zones principales de taille dynamique, la jeune génération (*young generation*) et l'ancienne ou vieille génération (*old generation*), étant chacune collectée par un algorithme différent. Une troisième zone est appelée la "chambre d'immortels" ("*permanent space*") et contient les objets permanents comme par exemple les objets `Class`, les méthodes et certaines structures propres à la JVM.

La jeune génération est constituée de l' "*eden*" et de deux "*survivor space*". L'*eden* est l'espace où sont alloués les nouveaux objets.



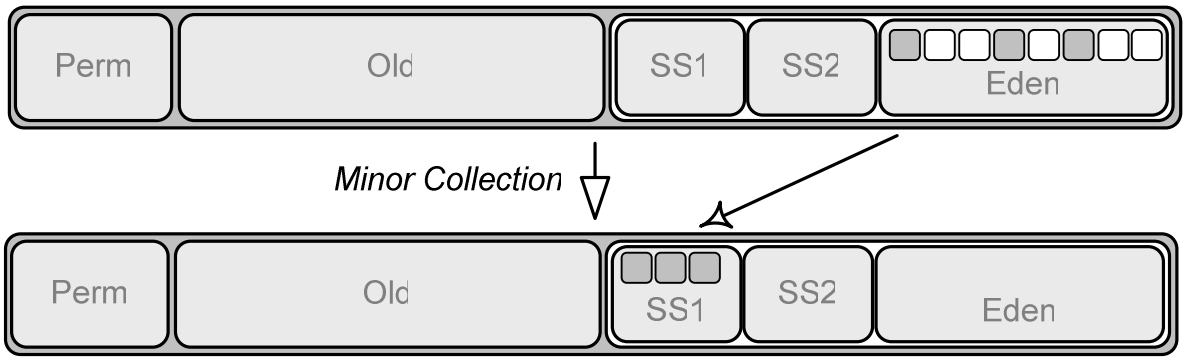
Découpage de la mémoire et des générations dans le JDK

Un des deux *survivor space*, que nous appellerons "from" sert à stocker les objets jeunes en transition et qui passeront éventuellement dans la vieille génération plus tard. L'autre que nous appellerons "to" est gardé vide jusqu'à l'exécution d'une *garbage collection* de la jeune génération qui est appelée "*minor collection*". Celle-ci se produit par exemple lorsque l'*eden* atteint sa capacité maximale de remplissage

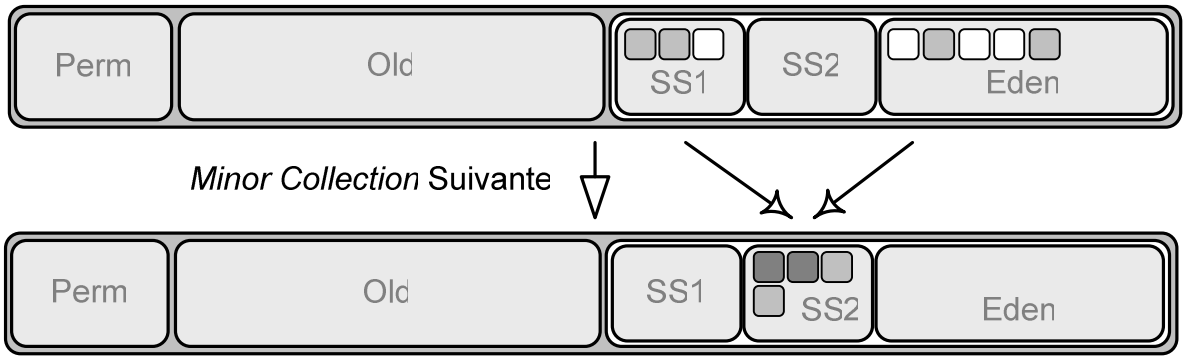
et utilise un forme d'algorithme de *copying garbage collection* qui copie les objets encore atteignables dans le *survivor space* "to". Après cela, l'*eden* et le "from" sont considérés comme vide et les noms et rôles du "to" et du "from" sont inversés. Cette méthode permet donc des allocations rapides en incrémentant simplement un pointeur dans l'*eden*, tout comme dans le *survivor space*.

Lorsqu'un objet a survécu à un certain nombre de *minor collections*, il est promu (*promoted* ou encore *tenured*) lors de sa dernière *minor collection* et est déplacé vers la vieille génération (Notons que certains objets sont parfois alloués directement dans cette génération, à cause d'une taille trop importante par exemple, et que d'autres objets peuvent être promus en avance parce que l'espace des survivants ne dispose pas d'assez de place).

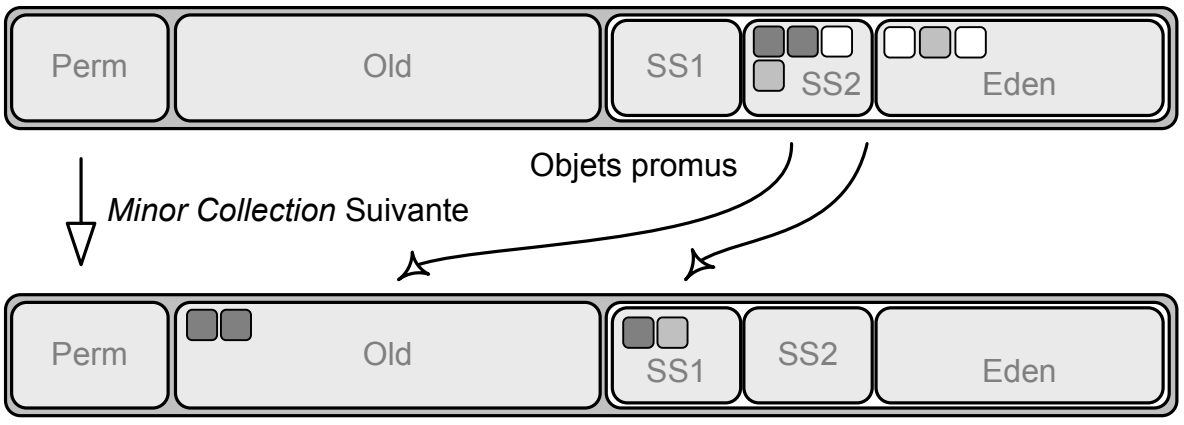
Les schémas à la page suivante nous illustrent les *minor collections*.



Lors de la première *minor collection*, les objets qui sont encore en vie dans l'*eden* sont copiés dans le premier *survivor space*, après quoi l'*eden* est libre pour de nouvelles allocations



Lors de cette seconde *minor collection*, les objets qui sont encore en vie dans l'*eden* et le premier *survivor space* sont copiés dans le second *survivor space*



Lors de cette troisième *minor collection*, les objets *tenured* (ayant survécu à deux minor collections en l'occurrence) sont promus vers la vieille génération

Illustration des Minor collections au sein du JDK
Inspiré du schéma "Minor Collections" du document [7]

L'ancienne génération est collectée lors d'une "*major*" ou "*full*" *collection*. Celle-ci se produit lorsque l'ancienne génération ou la génération permanente deviennent pleines, et toutes les générations sont collectées. En général, la jeune génération est collectée en premier avec son propre algorithme étant donné qu'il est plus efficace pour identifier les *garbages* dans cette génération. Après cela, l'algorithme de collection des anciennes générations est exécuté sur l'ancienne génération et la permanente. Notons que si l'ancienne génération ne contient pas suffisamment de place pour accueillir les objets qui vont probablement être promu lors de la collection de la jeune génération, l'algorithme des anciennes générations est utilisé sur le *heap* entier.

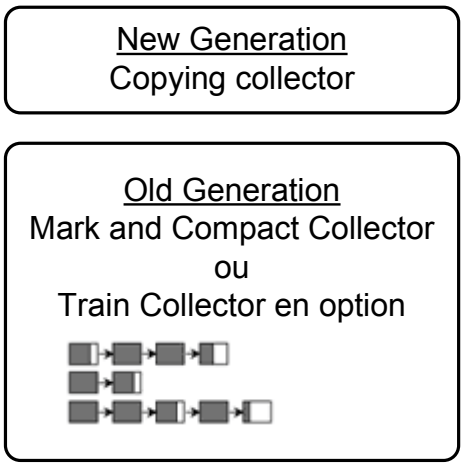
Cet algorithme de collection est de type *mark-sweep-compact*. La première phase identifie les objets atteignables. La seconde parcours les générations à la recherche de *garbages*. La troisième phase compacte les objets atteignables au début de l'ancienne génération et au début de la permanente en effectuant une compaction simple: une *sliding compaction* qui fait "glisser" les objets au début de la zone les uns après les autres.

Les *major collections* se produisent bien moins souvent que les *minor collections* mais prennent bien plus de temps. L'algorithme de *minor collection* est en effet prévu pour être le plus rapide possible étant donné qu'il est exécuté plus souvent, tandis que celui de *major collection* est prévu pour être efficace avec peu de mémoire libre et une faible densité de *garbages*.

10.1.2 Améliorations

L'arrivée de la version 1.2 du JDK a permis la diminution de la surcharge de travail due au *garbage collector* et la diminution du nombre de longues pauses de l'application en faveur de plus petites pauses pour le *minor collections*. Cependant, la combinaison du *garbage collector* générationnel et du *mark-compact garbage collector* ne résout pas le problème des longues pauses perceptibles dues aux *major collections*.

Une solution à ce problème est fournie avec une option qui permet de rendre les *major collections* incrémentales, éliminant ainsi les pauses trop longues au dépends du débit de l'application. L'algorithme utilisé est une sorte de *train collector* pour la collection de la vieille génération mais reste un *mark-and-compact* algorithme pour la génération permanente, ce qui est préférable étant donné que la collection incrémentale entraîne une surcharge de travail (maintenir à jour les références entres les trains/voitures). C'est à cause de cette diminution du débit de l'application que l'option incrémentale n'est pas activée par défaut. La *Hotspot JVM* commence en effet à s'orienter vers plus de débit pour l'application en diminuant la surcharge de travail de *garbage collection* pour favoriser les applications serveur.



Garbage collection au sein du JDK 1.2

10.1.3 JDK 1.4.1 & *Tuning*

La version 1.4.1 du JDK continue à viser les performances pour des applications serveur. Les nouveaux algorithmes sont en effet basés sur l'observation que beaucoup de machines utilisées pour des applications nécessitant de courtes pauses ou un haut débit possèdent souvent une grosse quantité de mémoire et plusieurs processeurs. Ces algorithmes sont donc optimisés pour tirer parti des ressources supplémentaires.

Cette version de la JVM offre de nouvelles options apportant ainsi de nouvelles possibilités au tuning du *garbage collector*.

10.1.4 *Garbage collector tuning*

Le *tuning* du *garbage collector* consiste ici à configurer des options de la JVM qui influencent le comportement du *garbage collector* dans le but d'améliorer les performances d'une application. Ces options peuvent être passées en ligne de commande lors du lancement de l'exécution d'une application par la JVM.

Quelques exemples d'options

La première chose à faire si l'on veut optimiser la *garbage collection* pour une application est d'observer le comportement du *garbage collector* pendant l'exécution de cette application.

En lançant la machine virtuelle avec l'option `-verbose:gc`, la console affichera des informations sur la quantité de mémoire libérée et le temps que cela a pris lors de chaque cycle de collection.

La version 5.0 fournit aussi une application qui permet d'observer l'exécution d'une autre: la console JMX. Il s'agit de l'exécutable "jconsole.exe" se trouvant dans le répertoire "bin" du JDK. Pour pouvoir observer une application avec cette console, il faut avoir lancé la JVM avec l'option `-Dcom.sun.management.jmxremote`, qui active JMX. JMX signifie "*Java Management eXtension*" et est une API qui permet de surveiller des application

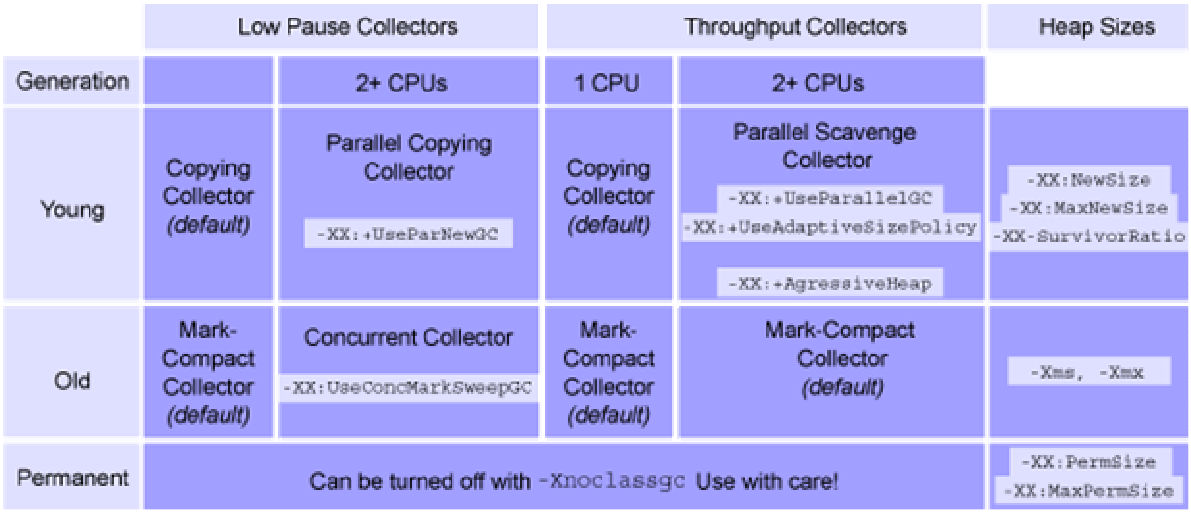
locales ou à distance. La console permet d'observer différents aspects de l'application comme par exemple l'occupation de la mémoire ou les *threads*.

Une première façon d'optimiser les performances du *garbage collector* est de choisir manuellement la taille des différentes zones du heap, ainsi que certains ratios entre elles. En voici quelques exemples:

Paramètre	Description
`-Xms`**n**	La taille initiale du *heap* = **n** bytes
`-Xmx`**n**	La taille maximale du *heap* = **n** bytes
`-XX:NewSize=`**n**	La taille par défaut de la jeune génération = **n** bytes
`-XX:MaxNewSize=`**n**	La taille maximale de la jeune génération = **n** bytes
`-XX:PermSize=`**n**	La taille maximale de la génération permanente = **n** bytes
`-XX:NewRatio=`**n**	Le ratio entre les tailles de la jeune et de l'ancienne génération = **n**. Par exemple, si **n** = 3, l'ancienne génération aura trois fois la taille de la jeune.
`-XX:SurvivorRatio=`**n**	Le ratio entre la taille de l'*eden* et d'un *survivor space* = **n**. Par exemple, si **n** = 3, l'*eden* sera trois fois plus grand qu'un *survivor space* et il occupera donc les 3/5 de la place de la jeune génération (car il y a deux *survivor spaces*).
`-XX:MinHeapFreeRatio=`**n** `-XX:MaxHeapFreeRatio=`**x**	Ces deux paramètres permettent de borner, par génération, le pourcentage d'espace libre avec un minimum de **n** et un maximum de **x**. Si après une *garbage collection*, ce pourcentage n'est pas entre les bornes dans une génération, la taille de la génération est augmentée ou diminuée de façon à ce que le pourcentage soit à nouveau entre **n** et **x**.

Le paramètre qui permet d'activer l'option des *major collections* incrémentales est `-Xincgc`. L'option `-Xnoclassgc` permet quant à elle de désactiver la *garbage collection* de la génération permanente (à ne faire bien entendu que si on est certain de ne pas la remplir).

Quelques nouveaux paramètres concernant les algorithmes de *garbage collection* sont apparus avec l'arrivée de la version 1.4.1 du JDK. En effet, avec cette version arrivent des algorithmes de *garbage collection* parallèles de façon à profiter de machine avec plusieurs processeurs. Certains de ces algorithmes parallèles vont favoriser la diminution des temps de pause (ce qui est le cas de l'option `-XX:UseParNewGC` pour la jeune génération et de `-XX:UseConcMarkSweepGC` pour l'ancienne), tandis que d'autres vont plutôt favoriser les débit de l'application (`-XX:UseParallelGC` pour la jeune génération). Cette dernière option est optimisée pour de grosses machines équipées de plusieurs processeurs avec un *heap* dont la taille se compte en *gigabytes*. Enfin, l'option `-XX:ParallelGCThreads=`**n** permet de choisir le nombre de *threads* de *garbage collection*. Ce nombre est par défaut égal au nombre de processeurs sur la machine. Notons enfin que toutes ces options sont destinées à être utilisées avec des machines à plusieurs processeurs et que leur utilisation avec des machines à un seul processeur pourrait diminuer les performances.



Options du *garbage collector* du JDK 1.4.1
"1.4.1 garbage collection options" provenant du document [5]

Toutes ces options permettent donc au programmeur d'essayer d'améliorer les performances de son application. Avec le JDK 5.0, si aucune contre-indication n'est donnée, la JVM essaye déjà de choisir elle-même certaines options optimales en fonction de la machine où elle se trouve en détectant par exemple si la machine fait partie de la classe des serveurs.

Malgré toutes les améliorations apportées à la JVM et au *garbage collector* au fil des ans, ces derniers ne suffisent toujours pas pour la programmation en temps réel. En effet, le *garbage collector* provoque toujours de (courtes) pauses *stop-the-world*, ce qui est inacceptable lors de la réalisation de *hard real-time systems*. De plus, même si de nouveaux outils ont été ajoutés au langage pour le support de contraintes temporelles, ceux-ci sont encore incomplets et insuffisants. Enfin, les améliorations se sont orientées vers des machines multi-processeurs avec beaucoup de mémoire vive et non vers des systèmes embarqués équipés de matériel de faibles performances.

11 Vers le temps réel en Java

Le document [8] présente nous explique qu'il existe différents types d'améliorations pour la gestion mémoire en temps réel pour le Java. Une première façon de faire est d'étendre le langage lui-même et de modifier la JVM. D'autres améliorations requièrent une assistance matérielle comme des processeurs créés spécialement pour exécuter directement des instructions du *bytecode* Java (par exemple le processeur *picoJava-II*). D'autres solutions enfin sont axées sur les systèmes d'exploitation.

La suite du présent document nous expose une solution du premier type complète pour la programmation de systèmes temps réel en Java: la *Real-Time Java Specification* et approfondit l'aspect gestion mémoire de celle-ci.

11.1 *The Real-Time Specification for Java (RTSJ)*

La RTSJ est une spécification qui fut créée en réponse à la JSR-1: la première JSR (*Java Specification Request*) du *Java Community Process* (JCP). La JCP est une organisation dont plusieurs entreprises du domaine Java sont membres et dont le but est de coordonner l'évolution du langage Java et des technologies qui lui sont associées.

"Temps réel" n'est pas synonyme de rapidité. Dans le contexte de la RTSJ, *real-time* signifie "la possibilité de répondre à des évènements du monde réel de façon fiable et prévisible". Donc, le temps réel est plus une question de *timing* que de vitesse et la prédictibilité est favorisées lors des compromis faits par la RTSJ.

Les classes de la RTSJ se servent du modèle Java comme base mais des changements fondamentaux doivent être apportés à la *Java Virtual Machine* pour certaines de ces classes.

La *Real-Time Java Specification* (RTSJ) qui nous est expliquées dans le document [1], améliore le Java dans les domaines suivants:

- gestion de la mémoire
- valeurs de temps et horloge
- objets *schedulables* et *scheduling*
- *threads* en temps-réel
- gestion d'évènements asynchrones et de *timers*
- transfert de contrôle asynchrone
- synchronisation et partage de ressources
- accès à la mémoire physique et *raw*

Notons que ces améliorations sont faites dans le cadre d'un seul processeur et qu'elles ne permettent pas, par exemple, de gérer l'allocation de *threads* à différents processeurs.

11.1.1 Gestion de la mémoire

Reconnaissant l'importance de la gestion mémoire, la RTSJ a tout d'abord requis que les *threads* en temps réel puissent prendre la place du *garbage collector* à l'exécution en un temps limité.

En plus d'un *garbage collector* pour le *heap*, la RTSJ utilise un système de gestion mémoire par régions. La RTSJ a donc introduit la notion de *memory area* (zone mémoire) pour qualifier les régions et la classe abstraite `MemoryArea` été créée pour représenter les régions mémoire. Toutes les régions mémoire de la RTSJ héritent de cette classe. Voici les différents types définis par la RTSJ:

- `HeapMemory`: permet l'allocation d'objets dans le *heap* standard du Java. Il s'agit de la seule région qui soit *garbage collected*. La RTSJ définit comment le programmeur peut interagir avec le *garbage collector* avec la classe abstraite `GarbageCollector`. Cette classe contient la méthode

```
public abstract RelativeTime getPreemptionLatency()
```

qui permet d'obtenir un objet représentant le *preemption-safe point* du *garbage collector*. Le *preemption-safe point* est le temps qu'un "objet *schedulable*" (cfr. le point suivant: "Les `RealTimeThread`") peut avoir à attendre pour prendre la place du *garbage collector* à l'exécution.
Un objet `GarbageCollector` représentant donc le *garbage collector* actuel peut être obtenu à l'aide de la méthode

```
public static GarbageCollector currentGC()
```

de la classe `RealtimeSystem`.

- `ImmortalMemory`: les objets de cette zone sont partagés entre toutes les *threads* de l'application et ne sont supprimés que lorsque le programme se termine. Ils ne sont donc jamais sujets à la *garbage collection*.

Il n'existe qu'une seule instance de chacune des ces deux classes. Ces instances représentent chacune leur région respective et sont accessibles via leur méthode `instance()`.

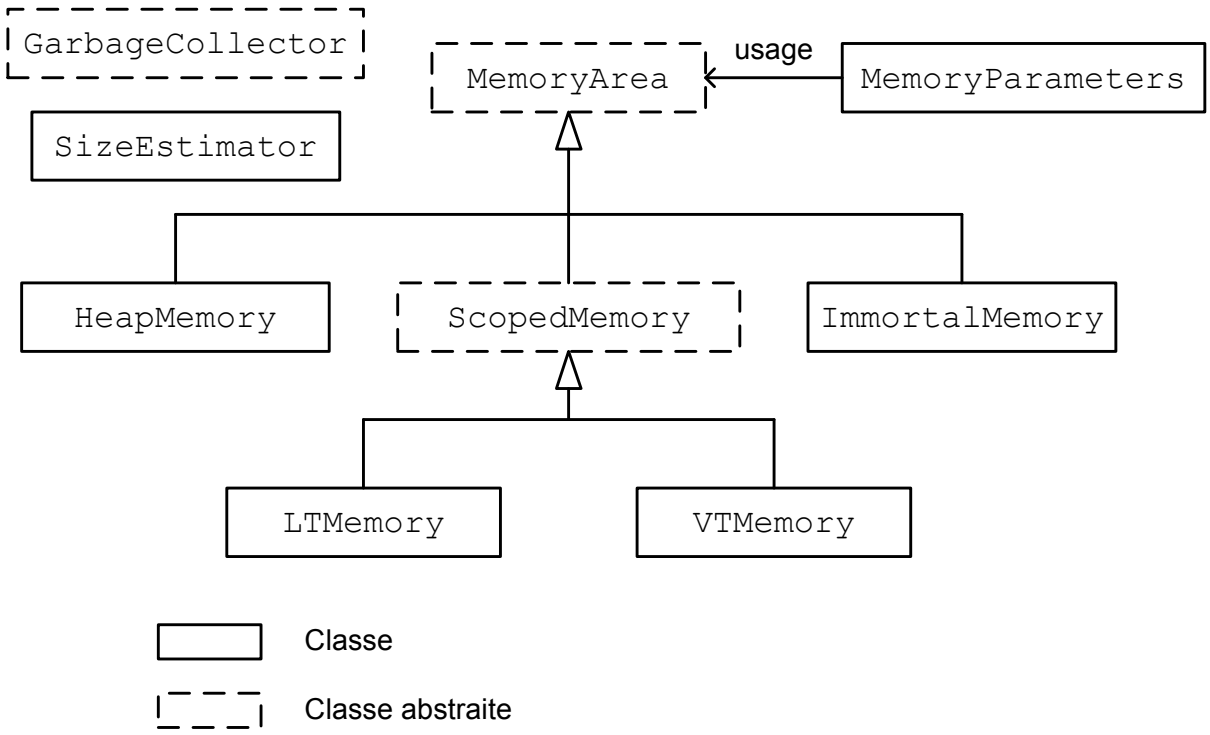
- `ScopedMemory`: il s'agit d'un type de mémoire où les des objets avec un temps de vie bien délimité peuvent être alloués. Ce type de mémoire peut être associé à des "objets *schedulable*" (cfr. le point suivant: "Les `RealTimeThread`"). Un compteur est associé à chaque région de ce type permettant de savoir le nombre de fois où des entités temps réel y sont entrées (et n'en sont pas encore sorties). Ce compteur et incrémenté (décrémenté) lorsqu'une entité temps réel y entre (en sort). Quand le compteur atteint zéro, la mémoire occupée par la région peut être libérée après l'exécution de la méthode `finalize` de chaque objet présent dans la région. `ScopedMemory` est aussi une classe abstraite dont les deux classe qui suivent héritent.

- `LTMemory`: l'allocation se produit en un temps linéaire (proportionnel à la taille de l'objet). Notons que l'on parle ici de l'allocation de mémoire à proprement parler, sans compter l'exécution du constructeur qui peut prendre un temps arbitraire.
- `VTMemory`: l'allocation peut prendre un temps variable. Ces régions sont prévues pour être moins prévisibles que les `LTMemory`, mais plus rapides.

Il peut exister plusieurs instances de `ScopedMemory` et celles-ci peuvent être imbriquées dans d'autres `ScopedMemory`.

Pour pouvoir estimer la taille d'une `ScopedMemory` en fonction des objets qui y seront alloués, la RTSJ fournit la classe `SizeEstimator`. Une instance de cette classe peut être utilisée pour évaluer la place que prendront des objets en mémoire sur base de la classe de ces objets. Cette instance peut être passée en argument au constructeur d'une `ScopedMemory` pour définir sa taille, mais comme son nom l'indique, il s'agit seulement d'une estimation. Celle-ci est juste un guide et n'est pas une valeur exacte: elle ne prend en effet pas en compte les objets qui sont créés lors de l'initialisation de l'objet à évaluer ou lors de l'exécutions de son constructeur. De plus, elle peut ou non tenir compte de la place prise par les *monitors*.

Il existe des règles d'assignation qui empêchent des objets contenus dans certaines zones mémoire de référencer des objets se trouvant dans d'autres zones mémoire à durée de vie plus courte pour éviter les *dangling pointers*.



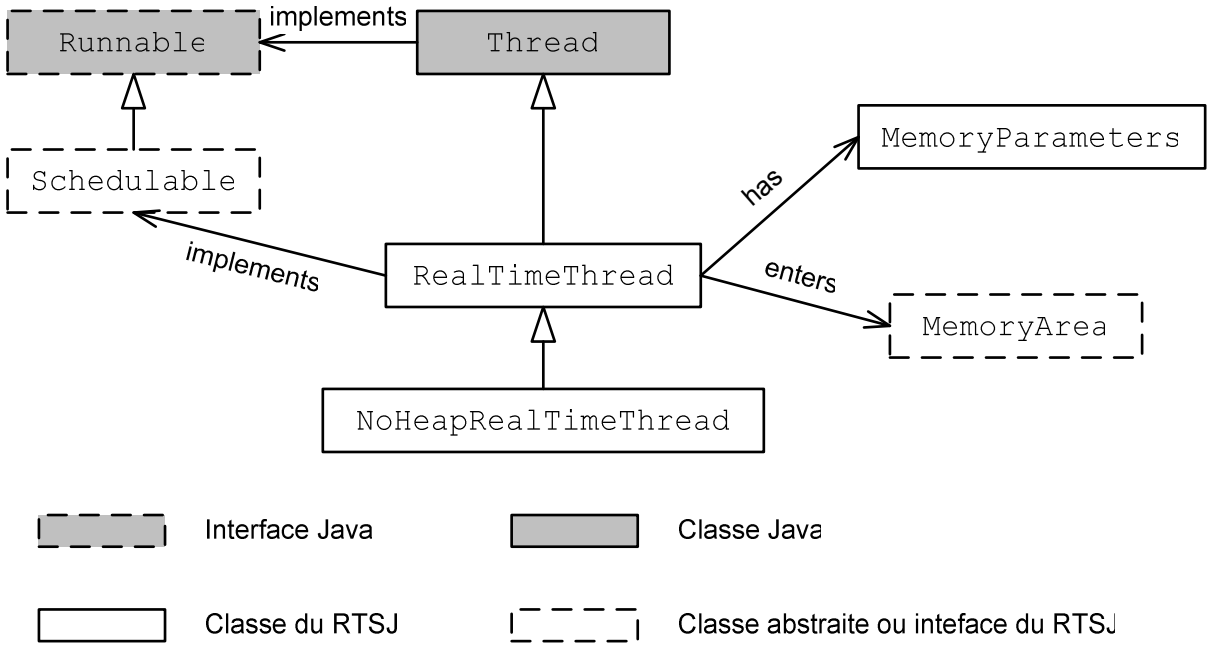
Hiérarchie des `MemoryArea`
Créé à partir du schéma *"Classes Supporting Memory Management"* de [1]

11.1.2 Les RealTimeThread

La RTSJ introduit de nouveaux objets appelés "objets *schedulable*": ce sont des entités temps réel telles que des gestionnaires d'évènements asynchrones ou des *threads* en temps réel (`RealTimeThread`). Les `RealTimeThread` héritent de la classe `Thread`.

Les `NoHeapRealTimeThread` héritent des `RealTimeThread` et sont des *threads* garantissent de ne pas créer ou référencer des objets dans le *heap*, rendant leur exécution indépendante du *garbage collector*. Toutes ces *threads* sont conçues et possèdent des fonctionnalités pour la programmation en temps réel.

Des paramètres concernant la mémoire peuvent être donnés aux objets *schedulable* même s'ils sont déjà actifs. Ils permettent de spécifier la quantité maximum que cette entité peut consommer dans sa mémoire par défaut, ainsi que celle dans l'`ImmortalMemory`. Ils permettent aussi de limiter le débit d'allocation du *heap* (en bytes par seconde). Ces paramètres pourront servir de base au *scheduler* pour une politique d'admission et/ou pour assurer une *garbage collection* adéquate.



Hiérarchie des `RealTimeThread`
Créé à partir du schéma *"Real-time Thread and its Parameters"* du document [1]

11.1.3 Utilisation des MemoryArea par les RealTimeThread

Les *threads* allouent par défaut de la place dans la `HeapMemory`. Pour que les objets créés par un objet *schedulable* soient alloués dans une `MemoryArea` choisie, la `RealTimeThread` peut "entrer" dans cette région. Pour ce faire, la *thread* peut appeler la méthode

```
public void enter(Runnable logic);
```

de la classe `MemoryArea`. Cette méthode reçoit un objet `Runnable` auquel la `MemoryArea` sera associé le temps de l'exécution de sa méthode `run`.

Par exemple, le code suivant nous montre la façon la plus simple d'allouer des objets dans l'`ImmortalMemory`.

```
ImmortalMemory.instance().enter(new Runnable()
{
     public void run()
     {
          // Toutes les allocations effectuées ici se
          // feront dans l'immortal memory
     }
} );
```

Il existe d'autres moyens d'allouer des objets dans une région en particulier. La méthode

```
public void executeInArea(Runnable logic);
```

de la classe `MemoryArea` permet elle aussi d'entrer dans la `MemoryArea`, mais se charge en plus d'exécuter le `Runnable` reçu en argument.

Pour allouer juste un objet dans une région donnée, les méthodes

```
public Object newInstance (Class type);
```

et

```
public Object newArray(Class type, int number);
```

de la classe `MemoryArea` permettent elles aussi l'allocation d'un objet ou d'un tableau d'objets dans la `MemoryArea` si le constructeur ne nécessite pas de paramètres. Si le constructeur requiert des paramètres, la classe `MemoryArea` propose la méthode

```
public Object newInstance(reflect.Constructor c,
Object[] args);
```

L'objet de type `Constructor` est un objet créé pour représenter un constructeur, tandis que le tableau d'objets représente les paramètres à passer au constructeur.

On peut par exemple créer un objet de type `T` dans la HeapMemory de la façon suivante:

```
Object o = HeapMemory.instance().newInstance(T.class);
```

Les *threads* classiques du Java ne sont en général pas autorisées à entrer dans une `MemoryArea`. Elles peuvent cependant appeler la méthode `executeInArea` sur l'`ImmortalMemory` et la `HeapMemory`. Elles peuvent aussi créer des objets dans ces régions via les méthodes `newInstance` et `newArray`.

11.1.4 Critique de la RTSJ

Le langage Java est incomplet pour permettre la programmation en temps réel. Il manque en effet des instructions permettant par exemple l'exécution d'une tâche à un moment donné ou en un délai déterminé (Celles-ci sont appelées des *real-time control facilities* et permettent au programme de se synchroniser par rapport au temps). L'un des principaux buts de la RTSJ était donc de palier à ce problème. Ces *facilities* sont fournies grâce à l'ajout de nouvelles classes (comme les objets *schedulable*) et à la modification de la machine virtuelle. Grâce à cela la RTSJ est capable d'imposer des contraintes de temps réel que le langage Java de base ne permet même pas d'exprimer. Enfin, comme c'est nécessaire lorsque l'on parle de programmation en temps réel, la RTSJ offre un environnement d'exécution prévisible.

Notons aussi que la RTSJ est compatible avec le langage Java. La RTSJ est en effet prévu pour pouvoir exécuter un programme écrit en Java classique.

Gestion Mémoire

Lorsqu'on parle de programmation en temps réel en Java, la gestion mémoire est au cœur du problème. Malgré toutes les découvertes faites pour améliorer les *garbage collectors* et résoudre les problèmes qu'ils causent, les *garbage collectors* ne convenaient toujours pas pour la programmation en temps réel en Java. Et même si le mieux eut été d'arriver à laisser la gestion mémoire complètement automatique, la RTSJ a choisi comme compromis d'utiliser la gestion mémoire par régions pour pouvoir offrir certaines garanties requises par le temps réel.

Le *garbage collector* est toujours présent pour la gestion mémoire du *heap*, pour des tâches n'ayant pas de contraintes de temps réel. Mais lorsqu'une tâche nécessite des garanties, elle ne peut se permettre d'être perturbée par le *garbage collector* et se doit donc d'utiliser des régions mémoire.

L'inconvénient évident de ces régions est de ramener des complications au programmeur en lui confiant une partie du travail de gestion de la mémoire et les

règles d'assignation de pointeurs entre différentes régions mentionnées au point 11.1.1 rendent cette tâche d'autant plus complexe.

Cependant, la gestion mémoire par régions est quand même plus simple que la gestion manuelle de la mémoire. Elle permet aussi l'allocation linéaire de mémoire et la libération des objets contenus dans une région en une fois et peut donc ainsi améliorer les performances par rapport à la gestion manuelle ou à la *garbage collection*.

La gestion mémoire par régions a aussi l'avantage d'offrir la possibilité d'augmenter la localité d'objets liés les uns aux autres. Il s'agit donc là d'un bon compromis entre la *garbage collection* qui est inadaptée au temps réel et la complexité de la gestion manuelle de la mémoire.

Conclusion

Même si la RTSJ n'est pas la solution idéale au point de vue de la gestion mémoire (entièrement automatique), elle reste sans doute la meilleure solution pour la programmation Java en temps réel à l'heure actuelle et ce tout particulièrement pour le *hard real-time*. La RTSJ peut en effet fournir des garanties quant aux temps de réponse d'une tâche. Un petit outil est d'ailleurs disponible sur le site de la RTSJ à l'adresse http://www.rtsj.org/tools/analysis1_instructions.html pour calculer les temps de réponse d'une tâche grâce à une applet Java. C'est aussi sur ce site (http://www.rtsj.org) que l'on peut trouver la spécification elle-même ainsi qu'une implémentation de référence (*RI: reference implementation*). On y trouve aussi un *Technology Compatibility Kit* (*TCK*) qui est un test permettant de certifier d'une implémentation qu'elle est conforme à la RTSJ.

Enfin, comme le souligne le document [1] dans sa conclusion, la RTSJ n'est pas sans aucun problème. Dès sa première version, la RI a révélé des inconsistances et des erreurs dans la spécification, mais ces problèmes ont bien entendu été corrigées dans les versions qui ont suivi. De plus, la RTSJ va pouvoir continuer à corriger ses problèmes, à évoluer et à s'améliorer sous la coordination du *Java Community Process*.

11.2 Une implémentation de la RTSJ: *Java SE Real Time*

La première implémentation commerciale de la RTSJ est le *Java Real-Time System* (Java RTS) réalisé par la société Sun Microsystems qui est décrite dans le document [12]. Il s'agit d'une implémentation robuste qui permet de satisfaire les demandes de *timing* rigoureux pour des tâches critiques dans des applications en temps réel.

La première version de ce produit requiert au minimum le J2SE 1.4.1, le système d'exploitation Solaris 10 et un processeur de la famille SPARC. Des versions ultérieures supporteront cependant le J2SE 5.0 ainsi que d'autres plateformes (OS et matérielles). Le produit a été développé pour des systèmes à double processeur, mais peut aussi être exécuté sur des systèmes avec un seul processeur. Cela entraîne entre autres des temps de latence plus importants, mais reste toujours une solution efficace pour gérer les exigences temporelles.

Il s'agit d'un produit payant dont on peut obtenir la licence ou une version d'évaluation sur le site de Sun. Le produit, dont la version actuelle sortie en mai 2007 est la 2.0, apporte quelques améliorations considérables et supporte maintenant le J2SE 5.0 ainsi que la famille des processeurs *x86* et inclut un *real-time garbage collector* (RTGC) innovant.
Ce dernier réalise ses opérations de façon totalement concurrente (pas de phases *stop-the-world*) et les temps de latence tournent autour des cent microsecondes. Il est robuste en ce qui concerne la protection des *threads* critiques du reste. Enfin, il est "*autotuning*" pour améliorer sa facilité d'utilisation: il n'y a que quelques paramètres à régler.

Ce produit a été conçu pour respecter la RTSJ (qui a elle-même été créée et étudiée dans le but d'amener le langage Java dans le monde du temps réel) et aucun document ou évaluation trouvé ne signale un manque à cela. Un non-respect de cette spécification est très peu probable (étant donné que le Java RTS passe le TCK de la RTSJ mentionné ci-dessus) et serait un *bug* dans le produit qui serait sans doute rapidement corrigé. De plus la société Sun qui a créé ce produit est une entreprise ayant une bonne renommée mondiale et connaissant très bien le monde du Java. Enfin, du point de vue des performances, certains documents trouvés sur Internet font état des bonnes performances obtenues avec le système.

Il s'agit sans doute du meilleur (si pas le seul) produit actuel pour la programmation de systèmes temps réel fiables en Java et ce, au moins pour les *hard real-time systems*.

On peut trouver sur internet différents documents et articles ventant les mérites du produit Java RTS et de ses performances. En voici quelques-uns:
- http://www.financialrealtime.com/stocks/stock-market-news/news684267.html
- http://fr.sun.com/sunnews/events/2007/mar/techdays/presentations/JavaRTS_TechDays2007_public.pdf
- http://fi.sun.com/sunnews/events/2006/technical_breakfasts/presentations/HelsinkiWorkshopMay2006.pdf
- http://www.slideshare.net/jboutelle/just-a-test-16533/
- et bien entendu la présentation du produit sur le site de Sun: http://java.sun.com/javase/technologies/realtime/index.jsp ([12])

Conclusion

Les difficultés et problèmes lies à la gestion manuelle de la mémoire nous ont amenés à dire que la nécessité d'une gestion automatique de la mémoire augmentait avec la complexité des systèmes, même si ce type de gestion amène de nouveaux problèmes.

Nous avons découvert les problèmes liées à la *garbage collection* pour la programmation en temps réel. De nombreuses variantes de *garbage collectors* tentant de résoudre ces problèmes ont été présentées, mais même si la *garbage collection* a fait pas mal de progrès depuis ses débuts, elle ne convient toujours pas pour beaucoup de systèmes temps réel ayant des contraintes trop strictes.

Le Java étant un langage *garbage collected*, il ne convient pas pour la programmation en temps réel. Il a cependant lui aussi fait évoluer son *garbage collector* pour amélioration ses temps de réponse et ses performances dans le but de mieux convenir à la programmation en temps réel. Malgré cela, le langage Java est encore loin de convenir à la programmation en temps réel et ce d'autant plus pour des systèmes ayant des contraintes strictes à respecter.

De plus, la spécification du langage elle-même n'est pas suffisamment complète étant donné qu'elle ne contient pas les fonctionnalités (instructions) nécessaires pour exprimer bon nombre de ces contraintes.

La RTSJ a donc complété le langage Java pour palier à ce dernier problème. Mais pour permettre la programmation en temps réel, cette spécification devait aussi résoudre les problèmes inhérents à la *garbage collection*. Pour ce faire, la RTSJ introduit la notion de régions mémoire au sein de son modèle.

L'utilisation des régions mémoire est en effet suffisamment prévisible pour permettre d'offrir les garanties nécessaires à la programmation en temps réel. Cela permet donc de garantir aux tâches critiques d'un programme en temps réel le respect de leurs contraintes, mais la *garbage collection* n'est cependant pas exclue au sein de la RTSJ pour les *threads* ayant des contraintes plus souples.

L'implémentation du RSTJ réalisée par Sun Microsystems s'est avérée concluante quant à la possibilité de faire des systèmes temps réel, mais aussi en ce qui concerne ses performances. La RTSJ est donc une solution amenant le langage Java dans le monde de la programmation en temps réel.

La programmation en temps réel est, de par sa nature, plus complexe que la programmation classique. Elle amènera de toute façon un travail supplémentaire pour pouvoir exprimer les contraintes supplémentaires qu'elle engendre.

Le langage Java continue sans cesse d'évoluer, mais les solutions pouvant lui permettre à l'heure actuelle de convenir pour les temps réel demandent toute un compromis, que ce soit par exemple une assistance matérielle ou encore le fait de devoir gérer des régions en mémoire.

Bibliographie

[1] Andy Wellings, Chapitres 1-4, 7, 8, 12, *Concurrent and Real-Time Programming in Java*, John Wiley & Sons Ltd, England, pages 1-261, 2004

[2] Bill Venners, "Java's garbage-collected heap, An introduction to the garbage-collected heap of the Java virtual machine", http://www.javaworld.com/javaworld/jw-08-1996/jw-08-gc.html, dernière révision connue le 8/01/2006, (Premier accès le 23/03/2007)

[3] Brian Goetz, "Java theory and practice: A brief history of garbage collection", http://www-128.ibm.com/developerworks/java/library/j-jtp10283/index.html, dernière révision connue le 28/10/2003, (Premier accès le 24/03/2007)

[4] Brian Goetz, "Java theory and practice: Garbage collection and performance", http://www-128.ibm.com/developerworks/library/j-jtp01274.html, dernière révision connue le 27/01/2004, (Premier accès le 24/03/2007)

[5] Brian Goetz, "Java theory and practice: Garbage collection in the HotSpot JVM", http://www-128.ibm.com/developerworks/java/library/j-jtp11253/, dernière révision connue le 25/11/2003, (Premier accès le 24/03/2007)

[6] Greg Holling, "J2SE 1.4.1 boosts garbage collection, Three new algorithms target near real-time applications", http://www.javaworld.com/javaworld/jw-03-2003/jw-0307-j2segc.html, dernière révision connue le 03/07/2003, (Premier accès le 23/03/2007)

[7] Ken Gottry, "Pick up performance with generational garbage collection", http://www.javaworld.com/javaworld/jw-01-2002/jw-0111-hotspotgc.html, dernière révision connue le 1/11/2002, (Premier accès le 24/03/2007)

[8] Maria Teresa Higuera Toledano, *Solutions à la gestion mémoire pour systèmes Java temps réel*, Université de Rennes, Rennes, France, 2002

[9] Olivier Schmitt, "La gestion de la mémoire en Java", http://schmitt.developpez.com/tutoriel/java/memoire/, dernière révision connue le 19/11/2006, (Premier accès le 23/03/2007)

[10] Ravenbrook Limited, "The Memory Management Glossary", http://www.memorymanagement.org/glossary/, dernière révision connue le 4/12/2001, (Premier accès le 10/08/2006)

[11] Tony Printezis, "Garbage Collection in the Java HotSpot Virtual Machine", http://www.devx.com/Java/Article/21977/, dernière révision connue le 17 septembre 2004, (Premier accès le 24/03/2007)

[12] Sun Microsystems, "Java SE Real-Time", http://java.sun.com/javase/technologies/realtime/index.jsp, dernière révision connue en 2007, (Premier accès le 24/03/2007)

[13] Sun Microsystems, "Memory Management in the Java HotSpot™ Virtual Machine", http://java.sun.com/javase/technologies/hotspot/gc/memorymanagement_whitepaper.pdf, dernière révision connue en avril 2006, (Premier accès le 24/03/2007)

[14] Sun Microsystems, "Tuning Garbage Collection with the 1.3.1 Java Virtual Machine", http://java.sun.com/docs/hotspot/gc/index.html, dernière révision connue en 1999, (Premier accès le 23/03/2007)

[15] Steve Meloan, "The Java HotSpot Performance Engine: An In-Depth Look", http://java.sun.com/developer/technicalArticles/Networking/HotSpot/, dernière révision connue en juin 1999, (Premier accès le 24/03/2007)

[16] Sven Gestegård Robertz, *Flexible automatic memory management for real-time and embedded systems*, Department of Computer Science, Lund Institute of Technology, Lund University, Lund, Suisse, 20003

[17] Thomas Gil, "Tout ce que vous avez toujours voulu savoir sur les Ramasses-Miettes .NET et Java", http://www.dotnetguru.org/articles/GC/GC.html, (Premier accès le 23/03/2007)

[18] Tim Lindholm, Frank Yellin, "The JavaTM Virtual Machine Specification, Second Edition", http://java.sun.com/docs/books/jvms/, dernière révision connue en 1999, (Premier accès le 23/03/2007)

[19] Urs Hölzle, *A Fast Write Barrier for Generational Garbage Collectors*, Computer Systems Laboratory, Stanford University

[20] Wikipedia, "Dynamic memory allocation", http://en.wikipedia.org/wiki/Dynamic_memory_allocation, dernière révision connue le 13/03/2007 (Premier accès le 10/08/2006)

[21] Wikipedia, "Garbage collection (computer science)", http://en.wikipedia.org/wiki/Garbage_collection_(computer_science), dernière révision connue le 22/03/2007 (Premier accès le 9/08/2006)

[22] Wikipedia, "Java version history", http://en.wikipedia.org/wiki/Java_version_history, dernière révision connue le 6/03/2007, (Premier accès le 24/03/2007)

[23] "The Memory Management Reference: Beginner's Guide: Recycling", http://www.memorymanagement.org/articles/recycle.html, (Premier accès le 9/03/2007)